

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A275 515



THESIS

DTIC
ELECTE
FEB 10 1994
S E D

CONCEPT DEFINITIONS AND SECURITY
REQUIREMENTS OF A COOPERATIVE EXECUTION
ENVIRONMENT FOR DISTRIBUTED COMPUTING

by

Terence Edward Busmire

September 1993

Thesis Advisor:
Second Reader:

Timothy J. Shimeall
Roger Stemp

Approved for public release; distribution is unlimited.

94-04511



DTIC QUALITY INSPECTED 5

94 2 09 046

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<p>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</p>				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE September 1993		3. REPORT TYPE AND DATES COVERED Master's Thesis, August 1991 - September 1993
4. TITLE AND SUBTITLE Concept Definitions and Security Requirements of a Cooperative Execution Environment for Distributed Computing (U)			5. FUNDING NUMBERS	
6. AUTHOR(S) Busmire, Terence Edward				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-500			10. SPONSORING/ MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Unclassified/Unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) <p>The problem that this thesis addresses is how to utilize the idle machine cycles of a network of workstations in an efficient, secure manner to accomplish cooperative execution of computationally-intense processes, without adversely affecting the normal use of the network resources by interactive users.</p> <p>The approach taken is to model a supervisory system of processes capable of monitoring the execution of computationally-intense procedures on individual workstations, halting computation and yielding the workstation resources when required to allow direct access by interactive users. The supervisory system will allow computation to resume when user access has ceased. Consideration has been made to maintain the security of network resources accessed by both users and cooperative execution processes.</p> <p>The result is a specification of the requirements for such a system, validated through experimental implementation and operation on a sample application.</p>				
14. SUBJECT TERMS Cooperative, Security, Distributed, Network, Linda, Polite, PVM, Spec, Node, Node process, Assign node, Cooperative Execution Environment			15. NUMBER OF PAGES 98	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT	

Approved for public release; distribution is unlimited

**CONCEPT DEFINITIONS AND SECURITY REQUIREMENTS OF A
COOPERATIVE EXECUTION ENVIRONMENT FOR DISTRIBUTED
COMPUTING**

by

Terence Edward Busmire
Captain, United States Marine Corps
B.S., United States Naval Academy, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the


NAVAL POSTGRADUATE SCHOOL
September 1993


Author:


Terence Edward Busmire

Approved By:


Prof. Timothy V. Shimeall, Thesis Advisor


Mr. Roger Stemp, Second Reader


Dr. Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The problem that this thesis addresses is how to utilize the idle machine cycles of a network of workstations in an efficient, secure manner to accomplish cooperative execution of computationally-intense processes, without adversely affecting the normal use of the network resources by interactive users.

The approach taken is to model a supervisory system of processes capable of monitoring the execution of computationally-intense procedures on individual workstations, halting computation and yielding the workstation resources when required to allow direct access by interactive users. The supervisory system will allow computation to resume when user access has ceased. Consideration has been made to maintain the security of network resources accessed by both users and cooperative execution processes

The result is a specification of the requirements for such a system, validated through experimental implementation and operation on a sample application.

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
DATE	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I. INTRODUCTION	1
A. BACKGROUND	1
B. OBJECTIVES	1
C. SPECIFIC AREA OF RESEARCH	2
D. SCOPE, LIMITATIONS, AND ASSUMPTIONS	3
E. ORGANIZATION OF THE THESIS	4
II. PREVIOUS WORK	5
A. BACKGROUND	5
1. Characteristics of Interest	5
a. Distributed Methodology	5
b. Security	5
c. Non-interference	6
2. Scope of Research	6
B. SYSTEMS EXAMINED	7
1. Linda	7
2. Polite	9
3. PVM	10
C. SUMMARY	11
III. THEORETICAL MODEL	13
A. BACKGROUND	13
1. General Overview of Functionality	13
2. Implementation Factors	13
3. Modeling in Spec	14
B. DEFINITIONS	14
1. Basic Terminology	14
2. Working Concepts	15

C. CONCEPT OF DESIGN	16
1. Overview of Model	16
2. Components of the Model	17
a. Subunit	17
b. Network Nodes	17
c. Node Process	19
d. Assign Node	20
D. MODELING PROCESS	22
1. Overview of Procedures	22
2. Translation into Spec	23
3. Definition Modules	24
a. Cooperative_execution_environment	26
b. Node	31
4. Machine Modules	35
a. Assign_node	36
b. Node_process	38
c. Node	41
d. Subunit	43
E. SUMMARY	44
IV. IMPLEMENTATION	46
A. BACKGROUND	46
1. Objectives	46
2. Methodology	46
B. PROTOTYPE	47
1. Modifications	47
2. Sequence of Events	49
C. TEST and VERIFICATION	50
1. Test Environment	50

2. Test Description.....	51
3. Test Results	52
D. SUMMARY.....	54
V. CONCLUSIONS and RECOMMENDATIONS.....	55
A. OVERVIEW of RESEARCH	55
B. RECOMMENDATIONS.....	56
1. Applications of Research.....	56
2. Potential for Further Research.....	57
APPENDIX A. Specification for Concept Definitions of CEE.....	59
APPENDIX B. Machine Specifications for CEE in Spec.....	62
APPENDIX C. Source Code for Node Process.....	66
LIST OF REFERENCES.....	84
INITIAL DISTRIBUTION LIST	86

LIST OF TABLES

TABLE 1	RESULTS OF NODE PROCESS IMPLEMENTATION	52
TABLE 2	MEMORY AND CPU MEASUREMENTS	53

LIST OF FIGURES

Figure 1	- Modular Design of CEE	23
Figure 2	- Explanation of Spec Concept Inheritance	26
Figure 3	- Concept Specification of CEE	27
Figure 4	- Concept Specifications for Cooperative Execution, Computation, and Sub-unit	28
Figure 5	- Concept Specification of Assigned_task	29
Figure 6	- Detail of Exec_state Specification	29
Figure 7	- Detail of Read/Write Resource and Completion Specifications	30
Figure 8	- Detail of Read_to, Write_From, Written, and Read Specifications	31
Figure 9	- Specifications for Node and Node_process	33
Figure 10	- Specifications for Ident, Avail_proc, Avail_mem, Avail_node, Assign_node, and Local_network	34
Figure 11	- Specifications for Execution and its Component Concepts	35
Figure 12	- Machine Specification for Assign_node	38
Figure 13	- Machine Specification for Node_process	39
Figure 14	- Machine Specification for Node	42
Figure 15	- Machine Specification for Subunit	43
Figure 16	- Process Flow of Node_p	47

ACKNOWLEDGEMENTS

I would be no less than ungrateful if I failed to mention the many people who contributed in one way or another to producing this work. First and foremost, my wife Karen deserves much thanks for the support and understanding she provided over too long a period.

Tim Shimeall provided no small amount of guidance, insight, and cajoling, particularly in maintaining the focus of the modeling process. He possesses an endless reserve of patience, and is an authority on writing code in C. Roger Stemp provided helpful insights on security issues, and steadily reinforced the KISS (Keep It Simple, Stupid) policy during the writing process.

This work would not have been possible without the help of these fine people.

I. INTRODUCTION

A. BACKGROUND

Current and future computing requirements are becoming ever more complex. Scientists conducting research in weather modeling, materials analysis, artificial intelligence, and other areas are formulating computational problems requiring greater amounts of computing power. At the same time, declining research budgets demand that current computing resources be utilized to the greatest extent possible. Additionally, the present rate of technological advancement in the area of computer science doubles the amount of computing capability available approximately every two years [HENN91]. Taken together, these factors could result in a "class separation" in the research community. The dichotomy caused by budget-conscious organizations, utilizing aging resources, struggling to keep pace with state-of-the art organizations, could serve to stifle the rate of scientific development dependent on computational analysis. Distributed computing offers a means of sidestepping these roadblocks to research. Utilizing existing computer resources in aggregate, in a more efficient manner, will allow researchers to harness CPU cycles that would normally be wasted as idle time. A network of workstations could be used to solve computationally-intense problems that would normally require supercomputer capability. The result of such a distributed approach to computation would be beneficial in terms of cost effectiveness and increased computing capability. However, in order to realize these benefits, the capability to use the distributed computing resources as a conventional multi-user network, allowing for multiple user processes, filesystems, and varying levels of privacy, must be retained. Thus, a security requirement exists; specifically, distributed computation should not interfere with the normal users of the distributed resources.

B. OBJECTIVES

The purpose of this research is to describe and model a Cooperative Execution Environment (CEE), which will perform the distributed computation across a network of

workstations in a secure manner. A CEE can be considered to be a networked collection of workstations, not necessarily homogeneous, each of which can be utilized for a computation-intensive problem while it is not being used. The availability of each workstation is determined by whether the machine is currently considered idle. A monitor procedure maintains availability information for all machines on the network and assigns computations to each machine as available. Because the network must remain available to direct users of the machines, the CEE must be able to recognize when a user is attempting to access a machine. That machine, in turn, must be considered unavailable for use. If it is currently processing a portion of a distributed computation, then that processing must cease, in order to allow the user to have access to the machine. Likewise, the local file system of each machine must be considered to be inviolable, so as not to compromise the integrity and security of user data. A specification of the CEE will be explored first followed by an implementation of the design.

C. SPECIFIC AREA OF RESEARCH

The primary focus of this research will be the construction of a model of the CEE. To this end, this thesis will address the following questions:

- 1 What defines Cooperative Execution?
- 2 What is the appropriate design for a Cooperative Execution Environment?
- 3 What security requirements exist for a Cooperative Execution Environment?

The definition of Cooperative Execution, assumes that several other definitions are in place. For example, the definition of Cooperative Execution may be given as:

Computation performed using otherwise-idle computer resources in a manner that protects the privacy and availability of the performing computer, its attendant storage and of the results of the performed calculation.

This definition in turn requires that computation, otherwise-idle computer resources, privacy, availability, and attendant storage be defined in a precise manner using a formal specification language.

Finding a design that is appropriate for Cooperative Execution involves a balance of concerns. The design must be specific enough to be implemented using available hardware and software, yet general enough to support any computation. A three-process model is used to realize this balance.

Finally, the question of security in a Cooperative Execution Environment also requires development of formal definitions. For instance, the distinction between direct execution and indirect execution becomes vital to maintaining security for local filesystems, as well as for the distributed computations that will be performed using Cooperative Execution.

D. SCOPE, LIMITATIONS, AND ASSUMPTIONS

While the design of Cooperative Execution proposed here addresses a multi-node network of machines controlled through a single central monitor, the focus of the modelling effort will be the requirements for a single node. This will result in a *node process* that will run on a single workstation, with the capability to detect workstation idle time and user interaction, retrieve a job assignment from a central monitor procedure, and monitor and regulate the progress of that job.

The design of the central monitor procedure, or *assign node*, is not addressed by this research, except for the interface that will be required to interact with the node process.

The following assumptions are made in the development of this model:

- a) The individual job, or *subunit*, will terminate itself upon completion.
- b) Measures external to Cooperative Execution provide protection against covert channels.
- c) The assign node is assumed to have the capability to break a computational problem down into individual subunits, each of which will be assigned to a different node process.

For purposes of this research, human interaction will provide the services of the assign node.

E. ORGANIZATION OF THE THESIS

Chapter II of this thesis will address previous work that has formed a basis for CEE. A detailed examination of the theoretical model of CEE can be found in Chapter III. Chapter IV contains a description of the prototype and validation of the prototype using an example problem. Finally, Chapter V summarizes the research and provides recommendations based on the conclusions reached.

II. PREVIOUS WORK

A. BACKGROUND

1. Characteristics of Interest

The design of CEE is based on concepts derived from several other distributed computing models. In order to explain the design in a meaningful way, it is necessary to examine these models, with particular attention directed to the characteristics each shares with CEE. Additionally, an examination of the limitations of each with respect to CEE will prove useful. The primary areas of consideration in the design of CEE are:

- a Distributed methodology
- b Security
- c Non-interference

a. Distributed Methodology

Generally, distributed computation is accomplished using one of several common approaches. These include single-processor systems networked together; multiprocessor systems, or a combination of the two. To achieve parallelism, these methods may utilize either an operating-system-based approach, permitting a conventional application to be parallelized, or a language-based approach, producing a parallelized application that runs on a conventional operating system [LELE90]. Since the goal of CEE is to provide enhanced computing capability while maintaining economy of operation, without impacting significantly on the normal use of the system, it would seem that considering a multiprocessor or parallel operating system implementation would be counterproductive. Indeed, the enhanced computing capability would only be a benefit where such an implementation does not already exist. Thus, a design based on a network of single-processor system using a distributed-application approach is desirable.

b. Security

The use of distributed computation, particularly a networked implementation, introduces certain security requirements as design considerations. Among these, the most

important are integrity protection of the distributed processes from non-CEE processes, and protection of non-CEE processes, or user processes, from CEE processes. Since a distributed, networked design is dictated by the goals of CEE, the possibility of compromise of the required interprocess communication must be considered.

c. Non-interference

As previously mentioned, the consideration for a networked, distributed design is predicated by the design goals of CEE. Likewise, the consideration of non-interference is also levied by the design goals. Since CEE is to make use of distributed computation facilities while simultaneously allowing user access to those same facilities, the degree of interference between CEE and this user access becomes a prime consideration. Ideally, such interference should be minimized.

2. Scope of Research

With the above considerations in place, it is now possible to address the scope of the research effort, and the rationale for making use of the systems that are examined in this chapter. *In order to maintain a productive focus*, two primary goals have guided this research. The first is that the design of CEE should be tailored to meet the objectives. In other words, the design should allow a practical and effective implementation to be realized. This includes the ability to demonstrate measurable improvements in computing capability, protecting against interference with users, and maintaining the integrity of CEE and user domains. The second goal is that the design of CEE should allow an "affordable" implementation to be realized. Such an implementation would be characterized by the use of existing hardware (networked workstations), running on an existing operating system (such as a version of Unix). The use of heterogeneous or homogeneous processors should not impact the implementation nor the design effort significantly. A design that conforms to these two goals would provide the maximum benefit of increased computing capability at little or no cost increase.

The distributed models considered in the design of CEE are Linda, developed by David Gelernter of Yale University; Polite, originally developed by Timothy Shimeall at the University of California, Irvine; and the Parallel Virtual Machine (PVM), developed by Jack Dongarra and Team PVM at the Oak Ridge National Laboratory and the University of Tennessee. A brief explanation of each model follows.

B. SYSTEMS EXAMINED

1. Linda

Although categorized as a programming language, Linda is more accurately described as a set of objects and related operations. When added to an existing language, Linda produces a new language that is suitable for distributed programming [WHIT88]. It was first developed by David Gelernter at Yale University in 1982 [GELE82].

Linda employs the concept of "tuple-space" to achieve distributed computation. Tuple-space is a shared data space, which acts resembles an associative memory. Tuples, or collections of fields, are inserted and removed from this conceptual tuple space by sender and receiver processes. The tuple is the atomic unit of computation in Linda, rather than the byte, as used in conventional languages. Each field of a tuple has a type associated with it. Additionally, each field can be either formal or actual. Formal fields have a type, but no value. Actual fields have a value as well as a type. Processes communicate in Linda through "tuple-matching", based on field value and placement within the tuple. All processes that are part of the same Linda program have access to this common tuple-space, and thus to every tuple released into it. The idea of tuple-space allows sender and receiver processes to be decoupled both spatially and temporally. Since tuples are inserted and removed from tuple-space asynchronously, senders never block, while receivers may block as needed. Because of the uncoupled nature, the approach is especially useful for multiple client-server applications.[WHIT88]

Linda's tuple-space allows the modeling of both shared-memory and distributed-memory models, as well as the implementation of either. These are the two general

categories used to classify what Leler calls multiple-instruction, multiple-data (MIMD) parallelism. Tuples passed between processes can represent shared variables, as used in a shared memory model, or they can represent messages passed between processes, as used in a distributed memory model. Additionally, the flexibility of tuple-space affords many advantages for parallel programmers. For example, dynamic load balancing can be implemented quite easily with Linda, allowing the computational load of a parallel problem to be evenly distributed among the available multiple processors of a system [LELE90].

While Linda has been used for application-level programming, Kernel Linda has been designed to support system-level programming. This would allow the implementation of a parallel operating system methodology, rather than a parallel application methodology. It adds such features to the basic Linda as multiple tuple-spaces, and tuple-spaces used as fields in tuples.[LELE90]

A number of implementations of Linda have been achieved. The first was on the experimental S/Net, at Bell Labs, by Nicholas Carriero. Multiprocessor implementations include the Encore and Sequent architectures, as well as the Intel iPSC. These are all based on the C language [WHIT88]. Additionally, a "Linda machine" is currently being developed, with Linda operations implemented in hardware [GELE88].

Of particular interest to the examination of the design of CEE is the implementation of Linda on a Local Area Network (LAN) performed at Sandia National Laboratories [Whit88]. A variation of VAX C, called VAX Linda-C, is used to support multiple processes on a single node, as well as multiple nodes communicating across a single Ethernet. In addition to the VAX machines, VMS machines were added to the network through a DECnet, and UNIX machines were accessed through TCP/IP. The tuple-space of this implementation was distributed across all the machines on the LAN. This was done to improve performance, at the expense of reduced robustness .[WHIT88]

The capability to model distributed memory using the tuple-space of a Linda program contributes much to the design of CEE. The message-passing that occurs in tuple-space reflects the methods of interprocess communication required to implement CEE.

Indeed, the distributed nature of Linda outlines exactly the basis of distributed processing in CEE. Linda fails to provide all the capabilities required for CEE, however, in that Linda implementations must be dedicated to distributed processing. Non-interference with normal use of the network is not currently available for Linda. This may change, however, with the development of Piranha Linda, a variant which will dynamically search for idle processors for task allocation [MARK92].

2. Polite

Polite is a UNIX-based utility that provides supervision for long-running jobs. It is based on work originally done by Timothy Shimeall, at the University of California, Irvine. The purpose of job supervision is to allow interactive users access to the system while long-running jobs are in progress. Polite accomplishes this by executing the long-running job, and periodically checking the system for logins and logouts. A new login will cause Polite to send a STOP signal to the supervised job. Conversely, when all users have logged out, a CONTINUE signal will be sent to a job that has been previously stopped.[POLI88]

A supervised job is started through Polite's command-line parameters. It is given its own process group, and signals are sent through this group. Any changes to the process group by the supervised job or any descendants of the job will cause Polite supervision to fail. Logins and logouts are detected through examination of the system *utmp* and *wtmp* files. Changes in these files will flag Polite that a login or logout has occurred. Further investigation is then performed, as to which machine the action occurred on. If it is the current machine, the appropriate signal is sent to the supervised job.[POLI88]

Should the machine be free of interactive users, Polite will *fork()* a child process, and execute the supervised job through an *execvp()* call. The memory space of the child process is overwritten by the resulting job execution, which runs to completion. The only control maintained over the job is through signals to the process group. After the *execvp()* call, Polite calls *fork()* again, to move supervision to the background and return the console

to the owning user. The owning user is, by default, ignored by Polite, but this can be changed through a command-line switch. Once in the background, Polite enters a loop in which it periodically checks for logins or logouts, as well as the status of the supervised job. Through the loop, it will call *sleep()* for an interval based on the presence or absence of interactive users. By default, this interval is the same whether users are present or not, and can be changed through command-line switches.[POLI88]

The monitoring capability of Polite is a key to the non-interference requirement of CEE. The concept of login and logout checking is practical and addresses the issue of interactive users in a straightforward manner. Much of the implementation of Polite has been adopted for use in the node process of CEE, providing idle detection, non-interference, and process control. Polite does not provide distributed computation capability, as it is designed for use on a single machine. Security provisions for interactive user processes and supervised processes are also absent.

3. PVM

The Parallel Virtual Machine (PVM) is an ongoing research project developed at the University of Tennessee, under the sponsorship of the Science Alliance, a state-supported program, the U.S. Dept. of Energy, and the National Science Foundation. It is a software system designed to allow a collection of heterogeneous computers to be used together as a concurrent computational resource. The current implementation places few restrictions on the individual computers that may be used, allowing for shared and local-memory multiprocessors, vector supercomputers, graphics engines, and scalar workstations. A flexible variety of interconnections may be used, including Ethernet and FDDI.[BEGU93]

Since PVM utilizes heterogeneous machines, the capability exists to exploit the strengths of individual machines connected to the system. Consequently, users are allowed to control the execution location of specific computation components. A set of PVM library routines provides access to the PVM system for user programs written in C or Fortran.

Through these routines, the PVM system handles message routing, data conversion predicated by incompatible architectures, and other tasks in a way that is hidden from the user.[BEGU93]

A PVM system is composed of two parts. The first is a daemon that resides on each machine making up the virtual machine. This is called pvmd3. It is designed to be installed on any machine. When it is started on a machine, it will start itself up on all machines given in the initial user-defined configuration information. This initializes the virtual machine, and allows a PVM application to be started from any of the connected machines. The second part composing PVM is the library of PVM interface routines. Applications must be linked using this library.[DONG93]

The distributed message-passing capabilities demonstrated by PVM would work well for CEE, which has a requirement for a similar capability. Additionally, the fact that PVM is not limited to using homogeneous machines contributes much to the design considerations of CEE. The PVM daemon, pvmd3, models quite closely the capabilities and properties of the node process used in CEE; however, PVM does not implement the CEE requirement for non-interference. While the machines utilized in a PVM system are available to interactive users, no provision is made to allow unimpeded interactive use of these machines while the virtual machine is actually configured and running. The requirement for CEE must allow for this.

C. SUMMARY

The three systems examined have much to offer the design and implementation of CEE. Linda offers distributed message passing, as well as some capability for idle detection. PVM, like Linda, offers distributed message passing, as well as characterizes the concept of implementation envisioned for CEE. Polite, while not classified in the same manner as the other two systems, offers a good model for the node process of CEE. The capability to detect idleness, and the provision of non-interference, qualify Polite as the

most suitable starting point in the implementation. Further, the availability of source code for Polite allows rapid prototyping and validation of the design of CEE.

III. THEORETICAL MODEL

This chapter offers a detailed examination of the preparation of a model of the Cooperative Execution Environment. Development of the model is discussed as a set of definitions and functional descriptions following CEE specifications.

A. BACKGROUND

1. General Overview of Functionality

CEE must be able to provide effective distributed computation. The system must be designed such that an implementation does not require special or additional resources, such as multiprocessors or supercomputers. CEE must also be designed so as to allow normal interactive use of the networked resources to proceed unencumbered by the distributed computation processes. This includes yielding the processor to user processes when required, as well as respecting the boundaries of user and CEE system resources. The functionality requirements above are again driven by the goals of increased computational performance of current resources, and the economic benefits such a design would provide.

2. Implementation Factors

Based on the above considerations, the type of resources utilized to implement CEE would have a significant influence on its design. The use of networked single-processor workstations rather than multiprocessors, for example, would require that the model be designed such that communication of large amounts of data between individual nodes is not necessary. The requirement to maintain the availability of each workstation and the entire network for use by interactive users serves to reinforce this design stipulation. Additionally, the use of networked processors, possibly in use by interactive users, requires an additional level of redundancy designed into CEE, one that perhaps would not be required in a multiprocessor implementation.

3. Modeling in Spec

The formal specification language known as Spec has proven extremely useful in modeling CEE. Spec is used to describe the behavior of the abstractions of a software system and its interactions with other systems and the external world [BERZ88]. Although designed for application to large-scale systems, it is suitable for smaller applications, particularly parallel or distributed systems [BERZ89].

The primary reason for using Spec as a modeling tool has been its availability to the designer, as well as the designer's familiarity with the syntax. However, Spec also offers some unique features that have proven quite useful in designing CEE. The fact that Spec is based on an event model of computation, yet still allows timing constraints to be specified, lends itself well to the requirements for CEE. The use of predicate logic rather than algebraic specifications to define the behavior of individual modules, independently of their internal structure, provides flexibility in the design specification. In other words, a "black box" design approach has been used, which allows a "glass box" specification when required. This serves to allow a complete design specification, yet one that allows complexity to be circumvented when it would obscure the objective.[BERZ89]

B. DEFINITIONS

1. Basic Terminology

As a prerequisite to addressing the design of CEE's components, it is necessary to establish some baseline definitions of terms and concepts employed in the design. Several of these follow:

Computer:A device used for computing. For purposes of this research, the term can be taken to mean a machine consisting of at least one CPU, dedicated memory, dedicated mass storage, a console device for user I/O, and ports for communication with other computers.

User:A term describing any individual who makes use of a computer. Such use can be interactive, or non-interactive (such as batch processing), and interfacing

with the computer can be direct or remote. For purposes of this research, a user shall be considered responsible for any non-CEE processes executing on a computer.

Computation:Evaluation of an algorithm or heuristic on a set of data; most typically, one that is CPU intensive rather than I/O intensive.

Attendant Storage:The dedicated mass storage associated with a computer. Also commonly referred to as the "local" storage. Normally this is a fixed-disk attached to the computer.

File System:The method a computer utilizes to store programs and data in a non-temporary manner. This may be a directory structure, implementing a hierarchical tree architecture. When referring to a computer's file system, such storage is normally assumed to be located on the attendant storage, or local disk.

Security:A property of computer systems consisting of the characteristics of secrecy, integrity, and availability.[PFLE89]

Secrecy:A characteristic of a computer system that applies when the system's assets are accessible only by authorized parties. Read access only is authorized.[PFLE89]

Integrity:A characteristic of a computer system that applies when the system's assets can be modified only by authorized parties. This implies read and write access.[PFLE89]

Availability:A characteristic of a computer system that applies when the system's assets are made available to authorized parties.[PFLE89]

2. Working Concepts

In addition to the above terminology, the following concept definitions are necessary before discussing the design of CEE:

Direct Execution:Execution performed immediately by an interactive user with respect to a specific computer, either through console or remote interaction. It is generally related to the purpose for which the computer was acquired.

Otherwise-idle Resource:A computer resource, such as CPU time, memory, or mass storage, not currently required by any direct execution (interactive user or other process). Current time can be interpreted as any time between time $(t-w)$ and time $(t+s)$, where t is the current clock time, w is the time interval for a transition of the resource between busy and idle, and s is the time interval for a transition of the resource between idle and busy.

Cooperative Execution:Computation performed using otherwise-idle computer resources in a manner that guarantees the availability of the performing computer for direct execution, as well as the security and integrity of its attendant storage and of the results of the performed computation.

Read Resource:A computer resource allocated to a process for reading. The process has authorization to examine, but not modify, the contents of the resource.

Write Resource:A computer resource allocated to a process for writing. The process has authorization to examine and modify the contents of the resource.

C. CONCEPT OF DESIGN

1. Overview of Model

At the highest level, CEE must be capable of allowing one or more computations to be performed concurrently, each computation on a single machine, in order to achieve the benefits of distributed computation. At the same time, it must not interfere with the use of each machine's resources by interactive users. Furthermore, it must maintain the security of its own resources, while respecting the security of each machine's resources. A closer examination of this capability is in order.

2. Components of the Model

a. Subunit

Each computation should be a part of a larger computation; ideally, one that lends itself to being parallelized, or broken down, into *subunits* of computation. The individual subunits would each be processed on a separate single-processor machine. Each machine would be connected to a common network, so as to facilitate communication between the machines. This communication would be necessary because of the need to assign specific subunits to specific machines, provide input data to those subunits requiring it, and collect the results of the individual subunit computations.

Although distributed computation could be achieved without this communication, it would not be practical. An implementation of CEE without networking features would require manual operation of the concurrently operating resources. This would mean restricting the resources of CEE to a common physical location, or failing this, would require an increase in manpower to coordinate the use of machines separated physically. Execution of subunits on individual machines would have to be accomplished manually, as would collection of computation results. Such an implementation would be time-consuming and manpower-intensive, decreasing and possibly negating the value of the computational and economic benefits offered by CEE. Thus, a network implementation of CEE is desirable.

b. Network Nodes

The number of machines, or *nodes*, used is not allowed to influence the design of CEE. Since one or more nodes will be used in an implementation of CEE, and since each of these nodes may or may not be available to CEE for processing, it cannot be known how many nodes will be available to any CEE implementation. Likewise, the number of subunit computations derived from a single CEE computing job cannot be known, as the degree of parallelization possible will vary from one job to another. In the ideal case, the number of nodes available to CEE will be at least equivalent to the number

of subunit computations composing a job. This will result in a job computation time equivalent to the computation time of the longest-running subunit computation, plus the time incurred by network overhead. This relationship is expressed in Equation 1 below:

$$t_r = t_j - \left(\sum_{i=1}^{n-1} t_i \mid (t_i \leq t_{i+1}) \right) + (\epsilon \cdot n) \quad (\text{Eq 1})$$

Where:

t_r is the computation time for the parallelized job.

t_j is the computation time for the non-parallelized job.

n is the total number of subunits composing the job.

t_i is the computation time for subunit i .

ϵ is the additional time per subunit caused by network overhead.

Conversely, in the worst case, only a single node would be available, which would force sequential processing of all subunit computations. The result in this case would be no speedup, and could possibly increase the required computation time, due to network overhead, to the point where it would be greater than the computation time of a non-parallelized job running on a single node. This worst case computation time is expressed in Equation 2 below:

$$t_r = \left(\sum_{i=1}^n t_i \right) + (\epsilon \cdot n) \quad (\text{Eq 2})$$

Where:

t_r is the computation time for the parallelized job running on a single node.

n is the total number of subunits composing the job.

t_i is the computation time for subunit i .

ϵ is the additional time per subunit caused by network overhead.

c. Node Process

In order to allow an individual node to be accessed by interactive users as required, CEE must be capable of monitoring that node for the presence or absence of user processes. This includes detecting when a user attempts to access the node, and detecting when the node is no longer being accessed by users. Additionally, this detection capability must operate at all times, while the node is performing a subunit computation, and while it is not. Should a user attempt to access the node while a computation is not being performed, access should be granted, and CEE should recognize that node as being unavailable for use. Should a user attempt to access the node while a computation is in progress, CEE should recognize the attempt, stop the computation, relinquish the node's resources, and allow the user access. Again, the node should be recognized as unavailable. In both cases, once the user has completed accessing the node's resources, CEE should recognize that the node is again available for utilization. Any suspended subunit computation should now be resumed. In order to achieve this, some part of CEE must be dedicated to each node utilized. This dedicated part will be referred to as the *node process*.

A case is then made for a certain level of generality in the design of CEE. The node process must be sufficiently non-specific in design such that it can be replicated on each machine used in a CEE implementation. Furthermore, because the nature of subunit computations cannot be known, this node process must be generic enough to monitor a wide range of subunit computations. Indeed, it becomes desirable to design this node process aspect of CEE in such a way so as to be functionally uncoupled from the generic subunit computation.

Assignment of specific subunits to specific nodes would be required because the status of each separate computation would need to be updated in an orderly fashion at intervals. Additionally, it would add a feature of redundancy to the design. Consider the case in which no tracking of subunits to nodes is performed: since a job is broken down into some number of subunits n , then n computations must be performed before the job computation is complete. Each subunit can be uniquely identified, as can be the results of

each computation. Should subunits be assigned to nodes without tracking the assignments, it is possible that the same subunit could be assigned to multiple nodes. This could possibly result in erroneous output, as well as increase the amount of time required to complete the entire job. Tracking subunit assignments would then allow computation to be performed efficiently. It would also provide a measure of redundancy to the design of CEE. Should one of the nodes utilized for a subunit computation be accessed by a user for a prohibitively long period of time, or should it fail altogether, the results of the entire job would be delayed while waiting for that subunit computation to complete; an event that may never happen. If it is known which subunit was assigned to that node, it is possible to restart that subunit on another available node. This in turn would require some sort of reporting scheme, such that each node process would keep track of the current progress of computation of the subunit assigned to it, and report such status to a central collecting point upon subunit completion. Accomplishing this would require CEE to have the capability to uniquely identify each node being utilized.

Should a job parallelization result in subunits with particularly long-running computation times, the redundancy actions just described could result in excessive job completion times. Rather than restarting a subunit computation from the initial state upon relocating that subunit's execution, it would be desirable to maintain a "current status" for that subunit computation. This current status could then be utilized to restart the relocated subunit in "mid-stream," avoiding duplication of computation efforts somewhat. The same reporting scheme used for subunit completion could be used to report this state information to the central collecting point. Status reports would be made in accordance with a periodic schedule, either temporally-driven or event-driven.

d. Assign Node

The concept of a central controlling authority for assigning subunits to nodes and collecting computation results is a logical next step in discussing the design of CEE. Such a control process would need to be aware of how many and which nodes were being

utilized by an instance of CEE, as well as which nodes were available or unavailable for computation. It would have to know the number of subunits composing a job, assign those subunits to available nodes, and maintain a record of those assignments. A current state of execution would have to be maintained for each subunit, to facilitate efficient computation of that subunit in the event that it had to be relocated to another node. In order to recognize that a subunit computation might require relocation, the control process would need to maintain a signalling mechanism, such as a timer, for each subunit. This mechanism would be initialized upon assignment of the subunit to a specific node for execution, and would serve to notify the assign node that the subunit's node had experienced a failure, or was undergoing a prohibitively long user access. This would prevent the entire job from being stalled unnecessarily by any delayed subunit computations. Upon notification of such a delay, the assign node must be capable of reassigning that subunit to another available node, and restarting computation with the most recently reported execution-state information for that subunit. Additionally, the results of a completed subunit computation would have to be received by this control process, and stored for eventual compilation with all other subunit results.

A likely name for such a process might be *control process* or *assign process*. However, the name *assign node* is more appropriate. This is because it is desirable to design CEE such that the assign node process operates on a single node, or machine, dedicated for use as a CEE control device. In addition to the responsibilities of subunit assignment, result collection, and execution-state tracking, the assign node would need to provide for the security of CEE job computations. Specifically, this entails management of all read and write resources allocated to all subunit computations, as well as those resources used by CEE to carry out the support functions listed above. All operational resources required by CEE, such as node availability information, pending subunit computation information, and subunit execution-state information, would be maintained either in the memory or local storage associated with this single node. Allocation of subunit read and write resources would be accomplished at the time the assign node became aware of the subunit. Upon

assignment of the subunit to a node for computation, access to these resources would be enabled through communication between the assign node and the node process associated with that node. One of the benefits of this approach is that the subunit can remain a "black box" procedure; neither the assign node or the node process need have any knowledge of the details of a particular subunit's operation, yet the subunit can access all resources required. Another benefit of this approach is that the local storage of each node will not be accessed by CEE processes. Subunit computations will only access resources that have already been allocated for use; node processes and the assign node will utilize only the local storage of the assign node host machine, for "record-keeping" information storage.

D. MODELING PROCESS

1. Overview of Procedures

The functionality required of CEE described in the previous section has been modeled using the functional specification language Spec. The approach utilized has been a modular one, in keeping with the description of the three components of CEE: assign node, node process, and subunit. While the node process has been described as a single entity, the functions it performs can be classified into two distinct areas. Because of this, the actual design of the node process consists of two parts: the node process, and the node. Figure 1, an overview of the modular design of CEE, illustrates this distinction.

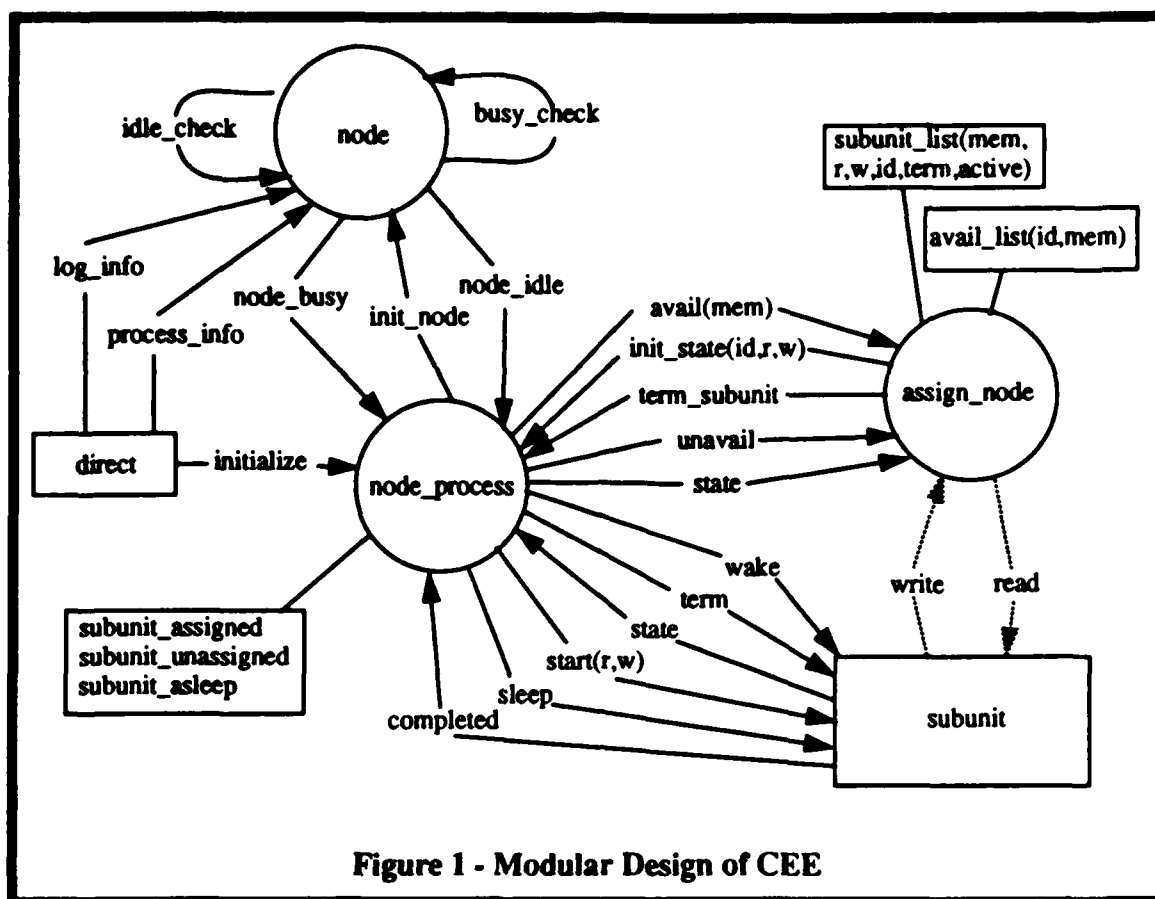


Figure 1 - Modular Design of CEE

The circles found in Figure 1 represent the components of the CEE model. The subunit portion of the illustration is shown as a rectangle because it is meant to be designed as a "black box" module. The line segments with arrowheads represent data flows, and are labeled appropriately. Rectangular fields connected to a CEE component by a line segment with no arrowhead represent memory states that a component must maintain. The dotted arrows between the subunit and the assign node represent controlled, secure access to read and write resources. The rectangle labeled "direct" represents direct access inputs.

2. Translation into Spec

Expressing the model of CEE described thus far using Spec closely models the diagram of the CEE design illustrated above. The entities shown in Figure 1: subunit, node,

node process, and assign node, are all treated as modules, the basic building blocks of Spec notation. Spec uses five types of modules: Functions, Machines, Types, Definitions, and Instances. Modules can be treated as having no internal structure, specified only through the interface with interacting modules. This interface consists of a series of stimulus-response events. Explicitly defined message transmissions are utilized to describe the set of stimulus events a module may receive. For each of these stimuli, another set of message transmissions is defined to describe the associated response of the module. The result of such a modeling scheme is that each module is independent of the internal structure of all others. This enables a top-down methodology that is quite appropriate to the design of CEE. [BERZ89]

CEE makes use of two types of Spec modules: Machines and Definitions. Originally, attempts were made to model some components as Spec Functions; these components were finally modeled as Spec Machines, which are mutable, when it became apparent that internal state information for those components would have to be maintained. Since this would require that the model account for some sort of memory capability, the Machine module was found to be better suited to this aspect of the design effort. The Definitions modules were necessary because several of the CEE components utilize shared concepts, which are not associated with any one component.

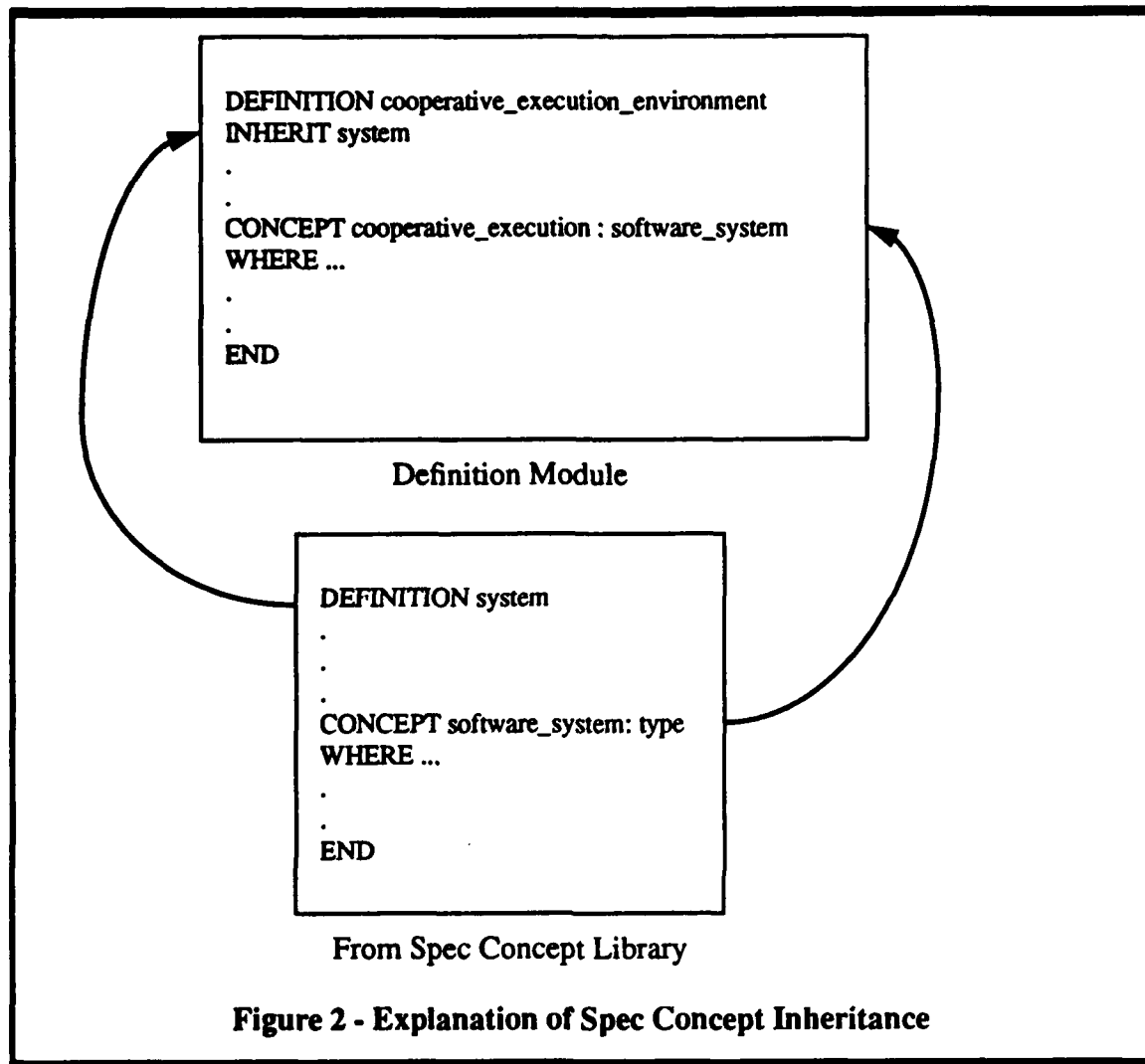
The specifications developed for the CEE model are provided in the following sections. In addition, an explanation of the functioning of the modules will be presented, as well as the rationale for specification decisions.

3. Definition Modules

The Specification for Concept Definitions of CEE, shown in Appendix A, describes the concept definitions that are required for the Cooperative Execution Environment, using the functional specification language Spec. These concept definitions incorporate Spec translations of the basic terms and working concepts shown previously in Sections B1 and B2 of this chapter. Definitions are given for the modules

cooperative_execution_environment and *node*. In accordance with Spec syntax, a specific definition is annotated with the keyword DEFINITION [BERZ88]. Each definition, in turn, consists of a series of concepts, denoted by the keyword CONCEPT. These concepts represent types and functions required to fully describe the system being defined. Generally, new concepts are defined using the predicate logic notation of Spec. Many concepts are pre-defined, however, and can be found in a standard library of Spec Concepts, located in Appendix C of [BERZ88]. A definition that makes use of this library, or any Spec concepts not defined locally, must use the keyword INHERIT, followed by the name of the definition module where the concept can be found. The results of such a scheme are increased productivity, reduced chance of error, and increased efficiency in module design.

As an example, the definition module *cooperative_execution_environment* inherits concepts from the modules *system*, *time*, *node*, *hardware_concepts*, and *system_actions*. All of these modules, with the exception of *node*, can be found in the Standard Spec Library. Definition module *node* is defined in Appendix A. Concepts defined in any of these inherited modules may be used in the definition of *cooperative_execution_environment*. For instance, at the highest level of definition module *cooperative_execution_environment*, the concept of *cooperative_execution* is defined to be of type *software_system*. *Software_system* is in turn a concept that is part of definition module *system*. An examination of Appendix A shows that definition module *system* is inherited by *cooperative_execution_environment*. This relationship is illustrated in Figure 2 below.



a. Cooperative_execution_environment

Definition module *cooperative_execution_environment* has concepts defined that are necessary to the proper functioning of CEE. These include the concepts of *cooperative_execution_environment*, *cooperative_execution*, *computation*, *assigned_task*, *subunit*, *write_to*, *written*, *read_from*, and *read*. The definitions of these concepts have been specified in terms of existing concept definitions that were previously defined in the

Spec Concept Library. The definitions concepts of *assigned_task*, *write_to*, *written*, *read_from*, and *read* are further specified using Spec predicate logic notation.

Concept *cooperative_execution_environment* is defined as a type of *software_system*, which, in turn, is a subtype of Spec type *system*. Essentially, it is the high-level components and sequence of events that compose the operation of cooperative execution. The specification, which is shown below in Figure 3, is annotated in terms of more specifically-defined concepts and modules. Although this results in a broad definition, the latitude exists for modeling a wide range of computational problems using *cooperative_execution_environment*.

```
CONCEPT cooperative_execution_environment : system
WHERE cooperative_execution_environment
=> assigned_task & local_network
-- define at high level the desired operation (assigned_task)
-- describe the sequence of events (globally)
```

Figure 3 - Concept Specification of CEE

Cooperative_execution and *computation* are both defined in a manner similar to *cooperative_execution_environment*. Both are of type *software_system*, a definition that provides a broad latitude of functionality. While *cooperative_execution* can be considered to be a high-level concept, *computation* is more a “building-block” concept. *Computation* is further used in the definition of concept *subunit*. Although *subunit* is also defined as a Spec machine module, the concept definition is necessary in order to properly define the security aspect of other concepts. The definition of a *subunit* concept as a type of *computation* also supports the model of CEE developed thus far, which depicts the subunit as almost an “atomic” operation or component of a larger task. As in the discussion of the design of CEE, the concept of *subunit* is sufficiently generic that a wide range of computational tasks could be used to meet the specification requirements. Figure 4 below shows these specifications.

CONCEPT *cooperative_execution* : *software_system*
– rules governing execution of operations in a
– *cooperative_execution_environment*

CONCEPT *computation* : *software_system*

CONCEPT *subunit* : *computation*
– required for characterization of security constraints.

Figure 4 - Concept Specifications for Cooperative Execution, Computation, and Subunit

Assigned_task, which has been referred to as job in the previous discussion of the development of CEE, is a concept defined to be of type *software_system*. It is specified as a vector entity, consisting of the components *exec_state*, *subunits*, *read_resources*, *write_resources*, *completion_criteria*, and *num_subunits*. These component concepts are further defined using predicate logic notation. The use of a vector in the definition of *assigned_task* is used to illustrate the association of many subunit computations composing a single job, or assigned task. Each subunit has associated with it a unique execution state, a vector of read resources, a vector of write resources, and a completion criteria met indicator. The completeness of an *assigned_task* is guaranteed, because the definition requires that the value of *num_subunits* be equivalent to the length of the vectors *subunits*, *read_resources*, *write_resources*, and *completion_criteria*; in other words, an *assigned_task* must be composed of at least one subunit, which must be complete. The specification for *assigned_task* is shown in Figure 5.

```

CONCEPT assigned_task : software_system
-- job to be supervised in the environment
WHERE assigned_task = (Exec_state, Subunits, Read_resources,
Write_resources, Completion_criteria,
Num_subunits) &
Num_subunits
    = Length(Subunits)
    = Length(read_resources)
    = Length(Write_resources)
    = Length(Completion_criteria) &
ALL(s in Subunits :: s : software_system) &
ALL(t : time ::
    ALL(e in Exec_state(t) :: e = (id, val))) &
ALL(r in Read_resources ::
    subset(r, union(files, devices, pipes))) &
ALL(w in Write_resources ::
    subset(w, union(files, devices, pipes))) &
ALL (c in Completion_criteria :: c: boolean)

```

Figure 5 - Concept Specification of Assigned_task

Exec_state is specified to be a time-dependent concept, such that each element of an *exec_state* vector is associated with an (*identifier*, *value*) pair. This is shown in Figure 6. Notice that no restriction is placed on the length of the vector, and none on the nature of the *identifier* or *value*. Also, although the reporting of execution state information by the subunit has been described previously as an event-driven occurrence, the *assign_node* associates the *exec_state* with a time value. This is because successive *exec_states* pertaining to the same subunit will be used to update the progress of the subunit's execution. The unique ranking of the time-stamp value allows this.

```

ALL(t : time ::
    ALL(e in Exec_state(t) :: e = (id, val))) &

```

Figure 6 - Detail of Exec_state Specification

Elements of the *read_resource* and *write_resource* vectors are defined as subsets of the union of all *files*, *devices*, and *pipes* allocated for reading and writing, respectively. The *completion_criteria* is defined as a boolean value, and serves to indicate whether the criteria required for a specific subunit completion has been met. These are detailed in Figure 7.

```
ALL(r in Read_resources ::  
  subset(r,union(files, devices, pipes))) &  
ALL(w in Write_resources ::  
  subset(w,union(files, devices, pipes))) &  
ALL (c in Completion_criteria :: c:boolean)
```

Figure 7 - Detail of Read/Write Resource and Completion Specifications

The concepts of *write_to* and *read_from* are defined in order to ensure that the security and integrity of read and write resources is maintained. These definitions state that the values of *write_to* and *read_from* will only be valid if a subunit attempts to write or read from a resource which has been previously allocated for that specific subunit. The actual write or read will not take place unless these values are valid. The concepts of *written* and *read* serve to indicate that a read or write has indeed occurred. Part of the requirement for this indicator is that the read or write is an authorized action for that particular subunit using that particular read or write resource. Figure 8 shows the specification for these concepts.


```

CONCEPT write_to(s : subunit, w : write_resource) :
  VALUE (b : boolean)
  WHERE b <=> SOME(i : integer, 1 <= i <= assigned_task.num_subunits) &
    s = assigned_task.subunits(i) &
    w = assigned_task.write_resources(i)

CONCEPT written(s : subunit, w : write_resource, d : data) :
  VALUE (b : boolean)
  WHERE b <=> write_to(s, w) &
    w = *w + d

CONCEPT read_from(s : subunit, r : read_resource) :
  VALUE (b : boolean)
  WHERE b <=> SOME(i : integer, a : assigned_task :: 1 <= i <= a.num_subunits) &
    s = a.subunits(i) &
    r = a.read_resources(i)

CONCEPT read(s : subunit, r : read_resource, rb : read_buffer) :
  VALUE (b : boolean)
  WHERE b <=> read_from(s, r) &
    subset(rb, r)

```

Figure 8 - Detail of Read_to, Write_From, Written, and Read Specifications

b. Node

Definition module *node* expresses several concepts that are derived from inherited modules *system*, *time*, and *hardware_concepts*. These include *node*, *node_process*, *ident*, *avail_proc*, *avail_mem*, *avail_node*, *local_network*, *assign_node*, *execution*, *supervised_execution*, and *direct_execution*. These concepts are grouped together into a single definition module because they are all based on the single node which has been described previously in the design of CEE. They may apply to a node utilized for subunit computation, as well as the node utilized to run the *assign_node*. As before,

specificity is used only as needed to attain the desired level of functionality, retaining the general applicability of the design.

Concept *node* is defined as type *processor*, inherited from definition module *hardware_concepts*. It is further specified that the *node* is completely developed, and that the *node_process* associated with the *node* can only be controlled through that *node*. Additionally, the *node* must be connected with the *assign_node* through a *local_network*. The concepts of *assign_node* and *local_network* are defined later in the specification. Note that there is no requirement for concept definitions concerning nodes which are not part of CEE. The requirements of concept *node* are sufficient to exclude any node which fails to meet the definition criteria. Also of note is that the concept of *node* is singular; nothing in the concept specification assumes or implies the existence of more than the single node being described, with the exception of the *assign_node*. This is significant, in that the node is considered to be an independent entity, unaware of the possible existence of other nodes, interacting only with the *assign_node*. The same observation can be made of *subunit*. Figure 9 shows the specification for concept *node*.

Node_process is a concept defined to be of type *software_system*, inherited from definition module *system*. It is specified in general terms, as a sequence of events, restricted to a *node*, which handles identifying idleness, receiving *subunit* computation assignments from the *assign_node*, interrupting the *subunit* computation if the processor becomes busy, resuming *subunit* computation when the processor is idle again, and routing *subunit exec_states* to the *assign_node*. A complete specification is given for *node_process* as part of machine module *node_process*. Figure 9 also shows the specification for concept *node_process*.

```

CONCEPT node : processor
WHERE
  completed(node) & controls(node, node_process) &
  connected_to((node, assign_node), local_network)

```

```

CONCEPT node_process : software_system
-- define that node processor handles identifying idle,
-- reports availability of node to assign_node,
-- accept subunit of assigned_task from assign_node,
-- interrupt subunit if processor becomes busy,
-- resume subunit when processor idle again
-- transmit execution_state to assign_node
-- sequence of events (per node)

```

Figure 9 - Specifications for Node and Node_process

Concept *ident* is defined to be of type *natural*, and is used to specify the concept of a unique identifier for each node utilized in CEE. Concept *local_network* is defined to be of type *network*, inherited from definition module *hardware_concepts*. It is used primarily in the definition of concept *node*, and requires no further specification. Concept *assign_node* is defined in a similar manner. It is of type *processor*, and is further defined in the machine module *assign_node*. Figure 10 shows the specification for these concepts.

The concepts of *avail_mem*, *avail_proc*, and *avail_node* are interrelated in that *avail_node* depends on the values of *avail_mem* and *avail_proc*. Concept *avail_node* is of type *processor*, with a boolean value. It is true for a specified node at a specified time, if *avail_proc* and *avail_mem* are true for the same node at the same time. *Avail_proc* is in turn a boolean which is true for a given node only if a computation is not being executed on that node at the time in question. *Avail_mem* is similarly defined as boolean. It results in a true value if and only if the sum of all computations executing on the node in question at the specified time occupy less space in the node's memory than the total memory available to the node. Figure 10 shows the specifications for these concepts, also.

```

CONCEPT ident(n : node) : natural
  where ALL(n1,n2:node :: (ident(n1) = ident(n2)) <=> (n1=n2))

CONCEPT avail_proc(n : node,t : time)
  VALUE (b : boolean)
  WHERE b <=> NOT(EXISTS(c : computation :: execution(n, c, t)))

CONCEPT avail_mem(n : node,t : time)
  VALUE (b : boolean)
  WHERE b <=> SUM(ALL(c : computation, location(c,t)
                    = n :: size(c, exec_state(t))) < memory(n)
CONCEPT local_network : network -- standard Spec concept

CONCEPT assign_node : processor -- master control for environment

CONCEPT avail_node(n : node, t : time) : processor --node is avail for computation
  VALUE (b : boolean)
  WHERE
    avail_proc(n, t) & avail_mem(n, t)

```

**Figure 10 - Specifications for Ident, Avail_proc,
Avail_mem, Avail_node, Assign_node, and Local_network**

It is of interest that the specification of the *avail_node*, *avail_proc*, and *avail_mem* concepts would allow an implementation of CEE to determine node availability based on the CPU load and memory capacity of a component machine. The implementation of the *node_process*, which is described in Chapter IV, does not take advantage of this opportunity, rather it uses an "all or nothing" strategy. However, the use of this method should not preclude future implementations from attempting to use CPU load as a measure of idleness, since this strategy results in could be even greater efficiency in the utilization of the resources available.

The definitions of concepts *execution*, *supervised_execution*, and *direct_execution* are specified in a similar manner. Concept *execution* applies to a specific node, a specific computation, and a specific time. It produces a boolean value of true if and only if a *direct_execution* or a *supervised_execution* is occurring. The concept of *direct_execution*, in turn, produces a boolean value of true if and only if the specified

computation is being processed on the node in question at the time of interest. This amounts to a process executing locally on the node. The concept of *supervised_execution* results in a boolean value of true if and only if the execution state of the *assigned_task* (job) is changing over a time interval, and the specific node of interest can be considered to be the location of the *assigned_task*. In other words, the subunits of the *assigned_task* are being processed, and since the location of the *assigned_task* at any time can be considered to be the set of nodes its component subunits are assigned to, the node has one of the subunits executing on it. Figure 11 shows the specifications for these concepts.

```

CONCEPT execution(n : node, c : computation, t : time)
  VALUE (b:boolean)
  WHERE b <=> direct_execution(n, c, t) | supervised_execution(n, c, t)

CONCEPT supervised_execution(n : node, a : assigned_task, t : time)
  VALUE (b:boolean)
  WHERE b <=> a.exec_state(t) /=
    a.exec_state(t-1) &
    location(a, t)=n

CONCEPT direct_execution(n : node, c : computation, t : time)
  VALUE (b:boolean)
  WHERE b <=> location(c, t)=n & processing(c, t)

```

Figure 11 - Specifications for Execution and its Component Concepts

4. Machine Modules

Appendix B contains the functional specifications for the Spec machine modules utilized in the design of CEE. These include modules *node*, *node_process*, *subunit*, and *assign_node*. A machine module is denoted by the Spec keyword MACHINE. A machine module represents an abstraction that is described in terms of a finite set of state variables. The keyword STATE is used to indicate the types of these variables. The keyword INVARIANT is used to indicate a constraint that must be satisfied in all reachable states of

the machine. The keyword **INITIALLY** denotes an initialization restriction that must be satisfied in only the first state. As with Spec definition modules, concepts can be inherited from other modules.[BERZ88]

The behavior of Spec machines is based on stimulus and response. The keyword **MESSAGE** denotes a message received by the machine, from some other module, or from the machine itself. Note that knowledge of the identity of the message originator is not required. Upon receiving a message, the machine may act in a variety of ways. A message may be sent to another module. This would be indicated by the keyword **SEND**. A reply to the originating module may be generated, denoted by the keyword **REPLY**. The received message may invoke a state transition, marked by the keyword **TRANSITION**. [BERZ88]

a. Assign_node

Machine module *assign_node*, found in Appendix B and in Figure 12, has one state variable, *j*, for *job* or *assigned_task*. The values it can assume are the *ident* values of each *subunit* in the sequence of subunits which compose it. Each *subunit* is then a state of the *assign_node*. The invariant constraint of true indicates that every reachable state must be a valid state. Initially, machine *assign_node* is restricted to non-empty jobs.

The receipt of an *avail* message will cause the *assign_node* to begin to step through its states, in order of *ident*. The variable *c* denotes the current subunit. If *c* is associated with a valid subunit, and that subunit is not currently active, and its memory requirements are less than or equal to the available memory being reported by the *node_process* in the *avail* message, then *assign_node* will send an *init_subunit* message to that *node_process*. *Assign_node* will then transition to the next state. Additionally, timing logic for the subunit will be initiated, to allow detection of the "loss" of the subunit through hardware failure or some other mishap.

An *unavail* message will not cause any actions to occur in the *assign_node*, with the exception of the initiation of additional timing logic, this time to guard against a

subunit being delayed an excessive length of time due to node unavailability. *Assign_node* will continue to monitor, waiting for the arrival of the next message.

The arrival of a *state(s)* message from the *node_process* can be handled several ways. If the execution-state information arriving with the message indicates that the subunit's *completion_criteria* has been met, the *assign_node* will reply with an *ack_complete* message. This "resets" the *node_process*, so that it may be assigned a new subunit if necessary. Should either the "lost_period" or the "too_long" timers have elapsed, the subunit should be considered lost or invalid, and the *assign_node* will send a reply of *term_subunit* to the *node_process*, essentially killing that computation, and resetting the *node_process*.

```

MACHINE assign_node(j : job)

  STATE (j: sequence(s:subunit))
  INVARIANT : true
  INITIALLY : j  $\rightsquigarrow$  [ ]

  MESSAGE avail
    FOR i = 1 TO count(s IN j)
      c = j(i)
      WHEN ~(c.active) & [c.mem <= avail.mem]
        REPLY init_subunit(c.id,r,w) TO node_process
        -- Also initialize timing logic for each subunit at this point.
        -- Should timer elapse before exec_state information is received,
        -- that subunit is considered "lost".
      END LOOP

  MESSAGE unavail
    -- if this is received from a node_process currently
    -- assigned a subunit, this will invalidate the "loss" timer
    -- mentioned above, but will start a new timer, to
    -- guard against excessively long unavailability.

  MESSAGE state(s)
    WHEN completed(s.completion_criteria)
      REPLY ack_complete

    WHEN period(lost_timer)
      REPLY term_subunit

    WHEN period(too_long)
      REPLY term_subunit

  END

```

Figure 12 - Machine Specification for Assign_node

b. Node_process

The specification for machine module *node_process*, found in Appendix B and Figure 13, has a state variable *s*, that can take on the values of *running*, *unassigned*, *asleep*, and *completed*. The invariant constraint again indicates that every reachable state must be a valid state, and the initial state is *unassigned*.


```

MACHINE node_process
  STATE s=(unassigned, running, asleep, completed)
  INVARIANT true
  INITIALLY s = unassigned
  MESSAGE initialize
    SEND idle_check TO node
    -- initialize node state variables, also called init_node in
    -- diagram.
  MESSAGE node_busy
    WHEN s = running
      SEND sleep TO subunit
      SEND unavail TO assign_node
      TRANSITION s = asleep
    WHEN s = unassigned
      SEND unavail TO assign_node
    OTHERWISE
      SEND unavail TO assign_node
  MESSAGE node_idle
    WHEN s = asleep
      SEND wake TO subunit
      SEND avail TO assign_node
      TRANSITION s = running
    WHEN s = running
      SEND avail TO assign_node
    OTHERWISE
      SEND avail TO assign_node
  MESSAGE init_subunit(r,w)
    -- Check state here and transition to assigned
    SEND start(r,w) TO subunit
  MESSAGE term_subunit
    -- Received from assign_node, after detection that
    -- this node's subunit has been lost from computation.
    -- Reports from this process will be irrelevant.
    -- Check state here and transition to unassigned
    SEND term TO subunit
    TRANSITION s = unassigned
  MESSAGE state(s)
    SEND state(s) TO assign_node
  MESSAGE completed
    -- received from subunit.
    SEND avail TO assign_node
    TRANSITION s = completed
  MESSAGE ack_completed
    -- received from assign_node
    TRANSITION s = unassigned
END

```

Figure 13 - Machine Specification for Node_process

Node_process receives many messages: *initialize*, *node_busy*, *node_idle*, *init_subunit*, *term_subunit*, *state(s)*, *completed*, and *ack_completed*. A review of Figure 1 will reveal that these messages correspond to the data flows depicted in the diagram. The first message shown in the specification, *initialize*, will cause *node_process* to send the message *idle_check* to *node*. This corresponds to checking to see if any users are accessing the node resources upon starting the node process on that node. Receipt of this message also results in the initialization of the state variables of the *node*, a message referred to in Figure 1 as *init_node*.

The *node_busy* message can have several effects, dependent on whether the current state of the *node_process* is *running* or *unassigned*. If the *node_process* is currently in a *running* state, the *node_busy* message will cause a *sleep* message to be sent to *subunit*, and an *unavail* message to be sent to *assign_node*. The state of the *node_process* will then transition to *asleep*. If the *node_busy* message is received while the *node_process* is in the *unassigned* state, an *unavail* message will be sent to *assign_node*. If the *node_process* is in any other state, a received message of *node_busy* will have no effect.

A *node_idle* message will similarly affect the *node_process*, depending on the current state. Receipt of the message when in the *asleep* state will cause a *wake* message to be sent to *subunit*, as well as an *avail* message to *assign_node*. A transition to state *running* will also occur. Should the *node_idle* message be received while the *node_process* is in the *running* state, no effect will be seen. If the *node_process* is in the *unassigned* or *completed* state, an *avail* message will be sent to *assign_node*.

The messages *init_subunit* and *term_subunit* will result in the *node_process* sending a *start* message or a *term* message, respectively, to the *subunit*. Transitions of the *node_process* will occur, to *running* or *unassigned*, respectively. Receipt of a *state(s)* message will cause the state information received with the message to be relayed to *assign_node*. A *completed* message will cause the *node_process* to send an *avail* message to *assign_node*. Finally, an *ack_completed* message will cause a transition from the *completed* state to the *unassigned* state, in effect "resetting" the *node_process*.

c. *Node*

Appendix B also shows the specification for machine module *node*. *Node* inherits concepts from the Spec definition module *period*, primarily because it uses the concept *phase* in determining the frequency of checks for machine idleness. The state variable of *node* is *s*, which represents a pair of values. The first is an indicator that can take on the values of *busy* or *idle*, while the second is a temporal interval value called *check_phase*. This represents the length of the time period between checks for machine idleness. The invariant constraint is that each reachable state be a valid state, and the initial state of the node is *<busy, time>*. The time component of this state is the current time, at initialization of the node. The node is assumed to be initially *busy* so as to provide for a maximum capability for non-interference. Figure 14 contains the specification code for *node*.

As with *node_process*, a review of Figure 1 will reveal a correspondence between the data flows depicted and the messages that *node* can receive, as shown in the specification. A message received of *idle_check* will cause *node* to check the information provided by *log_info*. If no users are accessing that machine, the *node* will then check the information provided by *process_info*. If the *percent_cpu* being utilized is zero, (aside from the cpu percentage used by the *node* and *node_process*), and the interval specified for a *busy_check_cycle* has elapsed, then the *node* will send *busy_check* to itself. It will also send a *node_idle* message to *node_process*. A transition of state will also occur, to a new state of *<idle, current time>*. If the *percent_cpu* utilized is not zero, then *node* will send *idle_check* to itself, and the procedure just explained will be repeated. If users are accessing the machine, a message of *idle_check* will also be sent to *node*.

```

MACHINE node
  INHERIT period
  STATE (s = [busy | idle], check_phase : time)
  INVARIANT true
  INITIALLY s = (busy, check_phase)
  MESSAGE idle_check
    access log_info
    WHEN users = 0
      access p_info(direct)
      WHEN percent_cpu = 0 and rem((time-check_phase),phase(busy_check))=0
        SEND busy_check TO node
        SEND node_idle to node_process
        TRANSITION s = (idle, time)
      OTHERWISE
        SEND idle_check TO node
    OTHERWISE
      SEND idle_check TO node
  MESSAGE busy_check
    access log_info
    access p_info(direct)
    WHEN [(users > 0) | (percent_cpu > 0)] and
      rem((time-check_phase), phase(idle_check))=0
      SEND idle_check TO node
      SEND node_busy to node_process
      TRANSITION s = (busy, time)
    OTHERWISE
      SEND busy_check TO node
END

```

Figure 14 - Machine Specification for Node

A message of *busy_check* will have similar effect. Receipt of the message will cause a check of *log_info* and *process_info*. If users are accessing the machine, or the *percent_cpu* utilized is greater than zero, and the interval specified for an *idle_check_cycle* has elapsed, *node* will send itself an *idle_check* message, and a *node_busy* message to *node_process*. A state transition to <busy, current time> will also occur. If a subunit is assigned to that machine, a *sleep* message will be sent to the *subunit*.

If no users are accessing the machine, or no other processes are executing on the machine, the *node* will send itself a *busy_check* message again.

A message of *init_node* will result in the initialization of the *node* state variables, to include *check_phase*, *idle_check_cycle*, and *busy_check_cycle*.

d. Subunit

The specification for machine module *subunit* is shown in Figure 15 as well as in Appendix B. While the *subunit* is considered to be a "black box", it still must conform to the design of CEE. This requires a certain structure be imposed on the component. *Subunit* utilizes a state variable *s*, that can take on the values of *run*, *sleep*, and *complete*. Each reachable state must be a valid state, and initially, the value of *s* is *run*.

```
MACHINE subunit
STATE s = (run, sleep, complete) -- also, task_specific info
INVARIANT true
INITIALLY s = run
MESSAGE start(r,w)
    SEND report to subunit
MESSAGE sleep
    SEND report to subunit
    TRANSITION s = sleep
MESSAGE wake
    TRANSITION s = run
MESSAGE term
    SEND report to subunit
    TRANSITION s=complete
    SEND completed TO node_process
MESSAGE report
    -- this is a temporal message. reports are made
    -- on a periodic basis to node_process.
    SEND state(s) to node_process
END
```

Figure 15 - Machine Specification for Subunit

A message received of *start* will cause *subunit* to send a *report* message to itself. This message, when received, will in turn cause a *state(s)* message, containing execution state information, to be sent to *node_process*. The frequency of the *report*

message is determined uniquely for each *assigned_task*. The inclusion of this frequency requirement is the responsibility of the programmer as he parallelizes the job into subunits.

A *sleep* message received will cause a *report* message to be sent to the *subunit*, as well as a transition of state to the value of *sleep*. Likewise, a message of *wake* will cause the *subunit* state to transition to *run*. Finally, receipt of a *term* message will cause *subunit* to send itself a *report* message once again, and then cause a state transition to *complete*.

E. SUMMARY

This chapter has examined in much detail the design and specification of the Cooperative Execution Environment. An overview of the design was first explored, progressing from a broad requirement, following an examination of suggested solutions, and culminating in the model depicted in Figure 1. Formal expressions of performance extremes were developed, demonstrating the best and worst case performance benefits that might be realized under an implementation of CEE. These were presented in Equations 1 and 2. The process of translating the model into a formal specification language was then explained, highlighting the interaction between components of the design.

This chapter has also served to clarify the definitions alluded to in Chapter I regarding the requirements for Cooperative Execution. Computation, defined in part B of this chapter, has now been specified as a Spec type of *software_system*. It is characterized at the lowest level as the *subunit*, and at the highest level as *assigned_task*. Otherwise-idle resources is a term used in the CEE model to describe machine resources that are not being accessed by an interactive user, and that have no processes executing at that time. The Spec concepts of *avail_proc*, *avail_mem*, and *avail_node* characterize this term. Privacy is provided for in the model of CEE by the concepts of *write_to*, *written*, *read_from*, and *read*. Availability of a node in CEE is determined using the concepts of *avail_node*, *avail_mem*, and *avail_proc*, and the concepts of *execution*, *direct_execution*, and *supervised_execution*. Attendant storage, defined to be the local storage of a machine, is protected from CEE use

by the vectors of *Read_resources* and *Write_resources* associated with the *subunits* of an *assigned_task*. With these requirements identified in terms of Spec concepts, the previous definition of Cooperative Execution can now be viewed as a more precise and exact specification.

The next chapter will present an implementation of the design specification. This implementation will be used in an attempt to demonstrate the validity of the design.

IV. IMPLEMENTATION

A. BACKGROUND

1. Objectives

The design of CEE has been expressed using a functional specification language, and being based on concepts shown to be valid through previous work, appears to be complete. As with any new system, though, the design validation would not be complete without a proper test of an implementation of the design. In addition to this validation, the implementation will demonstrate the feasibility of actual use of CEE in a productive manner.

2. Methodology

The prototype constructed for CEE is based largely on the Polite supervisor discussed in Chapter II. As mentioned, Polite's primary characteristic of interest is its implementation of non-interference with user access of the machine being utilized. This characteristic remains the primary interest in adapting Polite to use as the node process of CEE.

A complete implementation of CEE has not been realized, due to time constraints. However, the node process component has been implemented completely. This includes the node process proper, as well as the supporting structure for the subunit. Because the node process participates in all CEE interactions, the absence of the assign node component does not impact greatly on the implementation. Further, since all communication between the node process and the assign node is event based, much of the design of CEE can be validated using only the node process. For those events in which the assign node must participate, human interaction will suffice. In a complete implementation of CEE, such human interaction would not be required, but it is necessary for this test in order to achieve proper measurement and interpretation of results. Accordingly, all communication that would originate from or be received by the assign node has been implemented in the form

of ASCII text. The files used in this implementation are: "pending.subunits", which contains the subunit assignments made by *assign_node* to each *node_process*, and "node_status", which contains the availability of each *node_process*.

B. PROTOTYPE

1. Modifications

The implementation of the node process involves several modifications to the implementation of Polite. Figure 16 illustrates the flow of data in the final version of the node process.

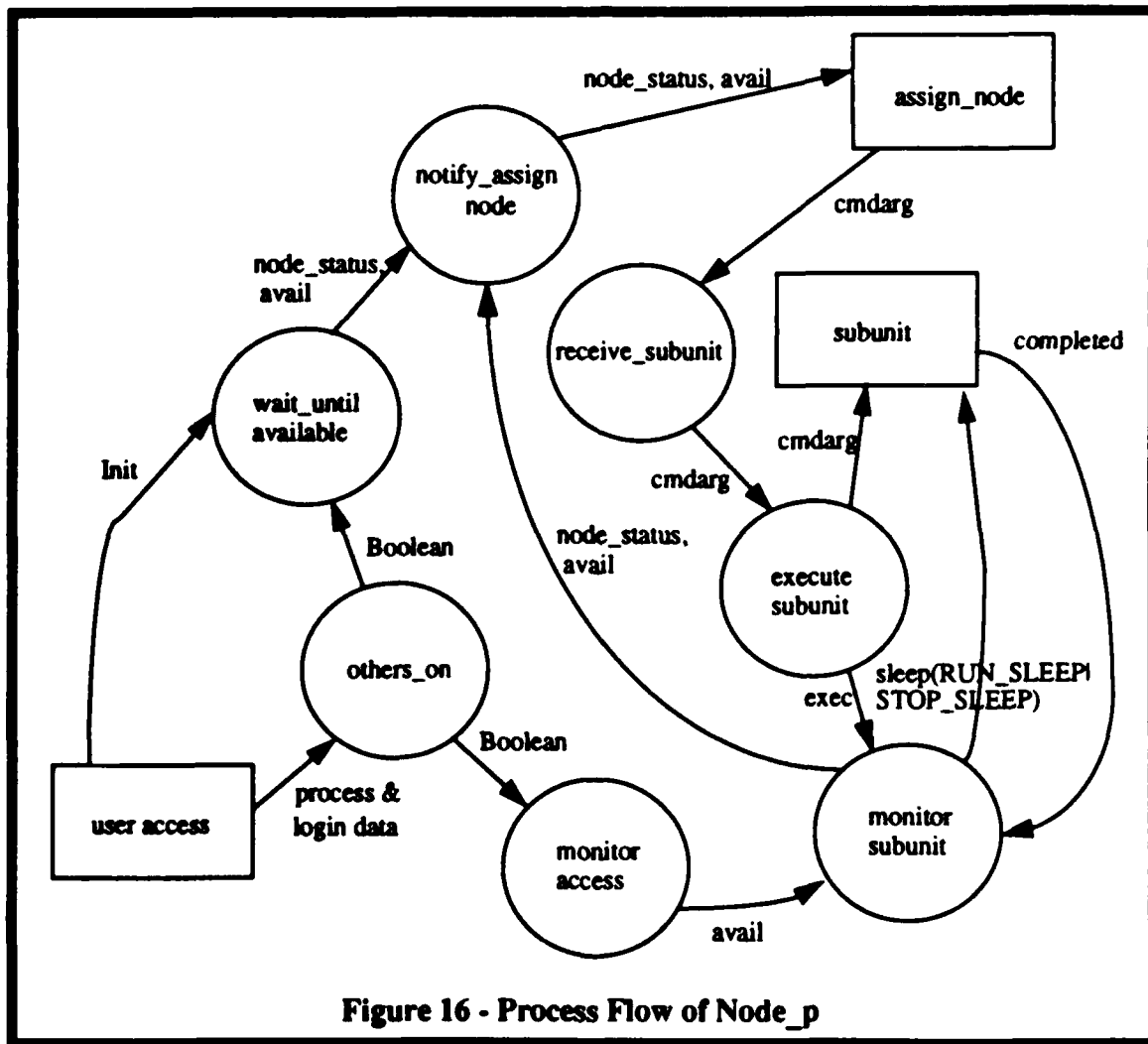


Figure 16 - Process Flow of Node_p

The modifications to the original Polite code are now described. The first of these is that input to the procedure is now taken from a text file, rather than the command line. This allows the human operator (performing the functions of the assign node) the capability to communicate with the node process in a manner similar to using a shared data structure, except that it is still accessible to humans. Additionally, the input to the node process must specify, in addition to the subunit procedure, specific read and write resources. This facilitates allocation of read/write resources, and is used to control access to these resources. Further, input to the node process is now uniquely assigned, by machine name. This is used to allow the "assign node" to track subunit assignments to available subunits. To support this, the node process now uses a *gethostname()* call immediately after initialization, and uses the returned string to report that node's availability or unavailability to the assign node. This reporting is accomplished again through a text file, to facilitate human operation.

The availability reporting is accomplished after first checking the local machine for the presence of user activity. This is accomplished in the same manner as the original Polite procedure, except that it is now accomplished immediately after initialization. Should a user be accessing the machine when the node process starts, the procedure will loop, invoking a *sleep()* call on each iteration. Once user access has stopped, the procedure will continue, reporting that node as available.

Polite originally made a *fork()* call to create a child process to run the client procedure, then called *fork()* again to move itself to the background, thus releasing the console for interactive use [POLS88]. Node process still accomplishes this, but it releases the console first, immediately after initialization. This is accomplished because the node process is designed to be a daemon, "immortal" while the workstation is in operation. The console must be available for interactive use, regardless of the presence of a subunit computation.

As mentioned above, the read and write resources are specified as input to the node process. The current implementation maps these resources to file descriptors five and

six of the client procedure. This requires that the client procedure be tailored to this scheme. It has been shown during development of the prototype that it is possible to map the read and write resources to the standard input and standard output of the client procedure. This demonstrates that special preparation of the subunit computation is not required, and that the node process can serve as a monitor for virtually any procedure. Further, the security mechanism designed into CEE is facilitated by having the node process control access to the read/write resources in this manner.

2. Sequence of Events

The node process prototype, upon startup, will initialize itself by checking for the existence of the operating system resources required for detection of login and logout. Failure of this check will cause the prototype to abort. This is acceptable, since without these resources, the operating cannot properly control interactive use. The prototype will also check for the existence of a file containing pending subunits. Should this file be missing, an abort will occur. The prototype will then determine the unique hostname of the machine it is running on. This will be used for informing the assign node of machine availability by name.

A *fork()* call is made, to move the process into the background, and release the console immediately for interactive use. Following this, the process enters an infinite loop. Required variables are initialized to default values, and another loop is begun, one which checks for interactive users accessing the machine. If users are present, the prototype will report itself as <UNAVAILABLE,UNASSIGNED> to the assign node, *sleep()* for a period of time (30 seconds in the test implementation), and repeat the loop. When the test for user access finally fails, the loop is exited, availability is reported as <AVAILABLE,UNASSIGNED>, and the file containing pending subunits is read. This file is a line-oriented text file, written out by the assign node (human operator). Each line begins with a machine name, and the node process will discard lines not matching the machine it is running on. Once a match has been found, the node process parses the line,

and retrieves the subunit process command, with any associated arguments that may exist. Also retrieved are the read resources and write resources allocated for the subunit. In the test, these are just file names, but in actual practice they would be data structures containing a vector of file or device names.

Once the appropriate line from pending subunits has been parsed, the prototype sends a change in availability status to the assign node of <UNAVAILABLE,RUNNING>. It then opens the read and write resources for use by the subunit, and calls *fork()* to create a child process. At this point, the child process has a copy of the parent's (node process) file descriptor table. Descriptors five and six are the read and write resources. An *execvp()* call is made in the child process, to start execution of the subunit.

Node process then enters the main subunit execution loop. A check is made again for the presence of interactive user access; if found, a STOP signal is sent to the process group of the child (subunit). The parent (node process) will then call *sleep()* for a period of time (30 seconds again). The loop returns. If no users access was found, a CONTINUE signal is sent to the process group of the child. Each time through the loop, a test is performed by the parent to determine if the child has completed execution. If it has, assign node is sent a new availability status of <AVAILABLE,COMPLETED>. The node process calls *sleep()* once more, and the availability status is reported again, this time as <AVAILABLE,UNASSIGNED>. In an actual implementation, the node process would retain the COMPLETED status until the assign node had "cleared" it from subunit assignment. The main loop then returns, and the node process resumes waiting for a new subunit assignment.

C. TEST and VERIFICATION

1. Test Environment

The actual test of the node process prototype was conducted on a network of four SPARC-1 workstations, Running SunOS version 4.1, a superset of BSD 4.3 [BSDM89]. The machines were designated *see1* through *see4*. One machine was used as the assign

node platform, meaning that the operator utilized this machine to monitor the test. The other three workstations each ran a copy of the node process. Remote logins to these three machines were performed from the assign node, as well as direct user logins on each machine. The operator accomplished assign node interaction with the three node processes through monitoring of text file output, and editing of text file input to the node processes. Additionally, the assign node platform was used to run the test procedure as a control measure.

The node process procedure itself, called *node_p*, was implemented in C. The primary reason for this choice was that Polite was written in C. It was also felt that this language would facilitate the low-level file structure manipulation that was required. While another language, such as Ada, would have allowed the implementation to more closely model the specifications proposed, the requirements would have led back to Unix system calls, which are best implemented in C.

2. Test Description

The procedure used to represent the parallelized job for this test is one known as *primetest*. It is written in C, and performs a test for primes over a range of integers. The range boundaries are specified as command line arguments. The first must be less than the second, and both must be no greater than 1.6 million. A read resource file, *prime.sub*, is utilized. *Primetest* expects this read resource to be mapped to file descriptor five. This is provided by *node_p*. Progress of the procedure's computation is reported on standard output.

Primetest was first run on a single workstation, using the node process on that workstation to monitor and control execution of the computation. No user access was performed on the machine during this run. The range used for this run was from 2 to 1600000, or the entire range. The next test utilized three workstations running the node process. On each machine, a subunit was assigned. In each case, the subunit was *primetest*, but operating on a subset of the entire procedure range of integers.

Execution time of the procedure for the entire range, as well as the sub ranges utilized were recorded. Additionally, process memory requirements and process CPU load for both *node_p* and the subunit *primetest* were recorded. The results are shown below.

3. Test Results

Table 1 displays the results of three tests of the *node_p* implementation, running *primetest* as a subunit. The input range of integers used for subunit is shown for each of the three machines which ran a portion of the test. The machine used to implement assign node functions, *see1*, shows results for the entire input range. Each test shows the execution time required for the subunit to process the input range of integers.

TABLE 1: RESULTS OF NODE PROCESS IMPLEMENTATION

		see1	see2	see3	see4
Test #1	Range	2 - 1600000	2 - 500000	500001 - 1000000	1000001 - 1600000
	Time	490 sec	115 sec	165 sec	235 sec
Test #2	Range	2 - 1600000	2 - 800000	800001 - 1200000	1200001 - 1600000
	Time	500 sec	195 sec	140 sec	160 sec
Test #3	Range	2 - 1600000	2 - 650000	650001 - 1150000	1150001 - 1600000
	Time	490 sec	148 sec	178 sec	178 sec

As can be seen for the results of test #3, *see2*, *see3*, and *see4* have completed processing the complete integer range of 2 to 1.6 million in 178 seconds. The same range of integers processed on *see1* took 490 seconds to complete. Thus, for test #3, which exhibits the closest equivalence of execution times, the implementation of node process on three machines completed processing of the entire range an average of 2.75 times faster than the processing of the same range on a single machine.

Table 2 shows the information gathered from a "ps" command during five tests of the node process and subunit on a single machine. Test #1 shows the measurements taken while the node process was available, with no subunit assigned, with no user accessing the machine. Test #2 shows measurements for the node process with a subunit assigned, with no user accessing the machine. A user access was in progress at the time of the measurements for Test #3. Both the node process and the subunit have been suspended, while the user was present. The measurements for test #4 were taken after the user access ended. Test #5 shows measurements taken after the subunit has completed processing. The node process is again available, with no subunit assigned.

TABLE 2: MEMORY AND CPU MEASUREMENTS

	Process	%MEM	SZ (kb)	RSS (kb)	TIME (sec)
Test #1	node_p	0.6	40	192	< 0:01
Test #2	node_p	0.7	40	212	< 0:01
	primetest	1.2	220	372	1:30
Test #3 (login)	node_p	0.0 - 0.3	40	0 - 100	< 0:01
	primetest	0.0	220	0	1:32
Test #4 (logout)	node_p	0.4	40	112	< 0:01
	primetest	0.3 - 0.7	220	80 - 220	8:00
Test #5	node_p	0.6	40	176 - 184	< 0:01

The column headings shown in Table 2 show the following information: %MEM- the percentage of real memory used by the process. SZ- the combined size of the data and stack segments. RSS- real memory size of the process. TIME- the CPU time used by the process at completion. [PSMA89]

The results shown in Table 2 reflect the expected values, for the most part. The measurements made for test #4, however, do not. It was expected that after the user access that was present during test #3 ceased, the measurements from test #4 would be similar to

the measurements of test #2. As the table shows, this was not the case. The measurements shown for test #4 are instead similar to those for test #3. It should be noted, though, that while both node process and subunit were suspended during test #3, they were not during test #4. The observed operation of the test does not agree with the measured values. Because this was a replicable phenomenon, and because the observed behavior of the prototype agrees with the expected behavior, this inconsistency in the measurements has been attributed to an extraneous "bug" in the *ps* command.

D. SUMMARY

The node process of CEE has been implemented and tested. The test procedures have shown results that are in agreement with those expected. The benefits of using existing equipment to gain improved computing power and efficiency have been demonstrated to be tangible and attainable. It is now time to examine what might be the most appropriate application of these benefits, as well as what direction future research in this area might take. Chapter V will present some thoughts on these subjects.

V. CONCLUSIONS and RECOMMENDATIONS

This chapter summarizes the contributions of this thesis, emphasizing the model of cooperative execution and the implementation of the node process. Recommendations for application of this work are presented, and areas of further research in this area are discussed.

A. OVERVIEW of RESEARCH

The definition of Cooperative Execution, at a broad level, is:

Computation performed using otherwise idle computer resources in a manner that protects the privacy and availability of the performing computer, its attendant storage and of the results of the performed calculation.

During development of the model and implementation of CEE, it became apparent that a host of issues are masked by the simplicity of this definition. Issues such as how to measure idleness; how to define privacy; how responsive must computing resources be to interactive use, and what is interactive use. Idle measurement was modeled and implemented using basic login and logout information. Privacy, or security, was dealt with at the file level, in terms of controlling all reads and writes of file resources through the node process. Resource availability and responsiveness to interactive use were modeled and implemented in a "hands-off" fashion; processing of the subunit computation and the node process was halted immediately when such use was detected, allowing full availability of the resources. The definition of interactive use was determined to be use of the resources by any non-CEE process; such as a user or user-process.

The appropriate design for CEE was developed through the model of CEE described in this work. The design developed here, using a single controlling assign node working in conjunction with one or more node processes, is an appropriate model for use on the target platform. Use of a node process on each machine of a network, reporting itself available or unavailable to the assign node as necessary, allows the most efficient use of the network resources, by both interactive users and CEE computations. Additional network overhead

is kept to a minimum, while the maximum number of machines cooperate in CEE. Tests have shown extremely low CPU usage for one monitor.

The security requirements of CEE are explored in terms of the broadest requirements and of some of the details provided in the implementation. The need to maintain the integrity of both CEE read and write resources, as well as the integrity of those resources belonging to other users of the system has been an issue of prime consideration. The conceptual design of CEE prevents CEE components from using local storage on host machines, while the functional specification defines *read_to* and *write_from* concepts that control access to machine resources. Future work in this area should explore these issues in greater detail, providing the security requirements for specific implementations of cooperative execution.

B. RECOMMENDATIONS

1. Applications of Research

A Cooperative Execution Environment would be useful in any situation requiring intense computational analysis. Such computation requiring processing of a large range of data would be particularly suited to the employment of CEE, since such problems are ideal for parallelization. Implementations of CEE need not be limited to networked workstations, either. A facility using nothing more capable than a network of desktop microcomputers would still be able to benefit from the capabilities of CEE. Some possible applications of CEE are presented below.

Military communications planning staffs would find CEE of great benefit. Communications planning routinely requires frequency management, as well as co-siting of all transmitters involved. For large military operations, the calculations can become quite formidable. Many times, these communications requirements must be performed by a high-level headquarters. The workload and complexity of the calculations is then multiplied by the number of subordinate headquarters involved. Additionally, timeliness of the results is vital. The ability to perform these computations in even half the current time,

while requiring virtually no additional hardware, makes CEE an attractive option to military communicators.

Military intelligence staffs could make use of CEE to speed processing of large amounts of data, particularly large amounts of textual information or imagery data. Message traffic analysis, performed on intercepted communications, could be performed on current equipment in less time. the same applies to computer enhancement of large satellite images, used for reconnaissance.

Intense numerical analysis, as required for weather and ocean simulation, could be performed by more facilities, using less-expensive, less powerful equipment. The result would be greater availability of interpreted data, which would speed research efforts tremendously.

2. Potential for Further Research

The modeling of CEE and the building of the node process prototype are just the first steps in developing a reliable tool for computational analysis. Future work in this area should start with an implementation of the assign node, and integration of it with the node process. The next step would be validation of the assign node and node process as a system against the functional specification of the model designed here.

Once these steps have been accomplished, performance enhancements could be explored. The current implementation utilizes an "all-or-nothing" approach to idle-detection on machines. It should be possible to expand this interpretation of idle-detection beyond the login and logout events, and take CPU-load into consideration. Eventually, a CPU-load threshold could be used as a measure of machine availability. This would allow even greater efficiency in the implementation of CEE.

In the same way, research may be done to determine when or if it is safe to utilize the local storage of individual host machines for CEE computations. This would involve detection of storage availability, which in turn would require its own definitions. It would also require a re-definition of the security aspects of CEE, particularly when sensitive or

classified information may be processed. The result would again be more efficient use of computing resources.

Incorporation of worm characteristics into CEE might be explored; the benefits of this approach to cooperative execution are unknown, and may prove to be greater than those of the current implementation. The capability to dynamically allocate workstation resources would again result in increased efficiency of resource utilization.

APPENDIX A. Specification for Concept Definitions of CEE

DEFINITION cooperative_execution_environment

-- concepts for describing cooperative execution and applicable environments

INHERIT system -- defines software_system, proposed, controls

INHERIT time

INHERIT node

INHERIT hardware_concepts

INHERIT system_actions

INHERIT requirement_goals

CONCEPT cooperative_execution_environment : system

WHERE cooperative_execution_environment

=> assigned_task & local_network

-- define at high level the desired operation (assigned_task)

-- describe the sequence of events (globally)

CONCEPT cooperative_execution : software_system

-- rules governing execution of operations in a

-- cooperative_execution_environment

CONCEPT computation : software_system

CONCEPT assigned_task : software_system

-- job to be supervised in the environment

WHERE assigned_task = (Exec_state, Subunits, Read_resources,

Write_resources, Completion_criteria,

Num_subunits) &

Num_subunits

= Length(Subunits)

= Length(read_resources)

= Length(Write_resources)

= Length(Completion_criteria) &

ALL(s in Subunits :: s : software_system) &

ALL(t : time ::

ALL(e in Exec_state(t) :: e = (id, val))) &

ALL(r in Read_resources ::

subset(r, union(files, devices, pipes))) &

ALL(w in Write_resources ::

subset(w, union(files, devices, pipes))) &

ALL(c in Completion_criteria :: c: boolean)

CONCEPT subunit : computation

-- required for characterization of security constraints.

CONCEPT write_to(s : subunit, w : write_resource) :

VALUE (b : boolean)

WHERE b <=> SOME(i : integer, 1 <= i <= assigned_task.num_subunits) &

s = assigned_task.subunits(i) &

w = assigned_task.write_resources(i)

CONCEPT written(s : subunit, w : write_resource, d : data) :

VALUE (b : boolean)

WHERE b <=> write_to(s, w) &

w = *w + d

CONCEPT read_from(s : subunit, r : read_resource) :

VALUE (b : boolean)

WHERE b <=> SOME(i : integer, a : assigned_task :: 1 <= i <= a.num_subunits) &

s = a.subunits(i) &

r = a.read_resources(i)

CONCEPT read(s : subunit, r : read_resource, rb : read_buffer) :

VALUE (b : boolean)

WHERE b <=> read_from(s, r) &

subset(rb, r)

END cooperative_execution_environment

DEFINITION node

-- concepts used to characterize the types of nodes

-- utilized in CEE, and the means to use them

INHERIT system

INHERIT time

INHERIT hardware_concepts

INHERIT requirement_goals

CONCEPT node : processor

WHERE

completed(node) & controls(node, node_process) &

connected_to({node, assign_node}, local_network)

CONCEPT node_process : software_system

-- define that node processor handles identifying idle,

-- reports availability of node to assign_node,

-- accept subunit of assigned_task from assign_node,

-- interrupt subunit if processor becomes busy,

-- resume subunit when processor idle again

-- transmit execution_state to assign_node

-- sequence of events (per node)

CONCEPT ident(n : node) : natural

where ALL(n1,n2:node :: (ident(n1) = ident(n2)) <=> (n1=n2))

CONCEPT avail_proc(n : node,t : time)

```

VALUE (b : boolean)
WHERE b <=> NOT(EXISTS(c : computation :: execution(n, c, t)))

CONCEPT avail_mem(n : node, t : time)
VALUE (b : boolean)
WHERE b <=> SUM(ALL(c : computation, location(c, t)
                = n :: size(c, exec_state(t))) < memory(n)

CONCEPT local_network : network -- standard Spec concept

CONCEPT assign_node : processor -- master control for environment

CONCEPT avail_node(n : node, t : time) : processor --node is avail for computation
VALUE (b : boolean)
WHERE
    avail_proc(n, t) & avail_mem(n, t)

CONCEPT execution(n : node, c : computation, t : time)
VALUE (b: boolean)
WHERE b <=> direct_execution(n, c, t) | supervised_execution(n, c, t)

CONCEPT supervised_execution(n : node, a : assigned_task, t : time)
VALUE (b: boolean)
WHERE b <=> a.exec_state(t) /=
    a.exec_state(t-1) &
    location(a, t)=n

CONCEPT direct_execution(n : node, c : computation, t : time)
VALUE (b: boolean)
WHERE b <=> location(c, t)=n & processing(c, t)

END node

```

APPENDIX B. Machine Specifications for CEE in Spec

--*** Machine Specification for CEE ***

- Assumptions:
- Precheck for covert channel in subunit (other than R/W)
- Assign guarantees access to (R/W)
- Pre-know max mem size for subunit
- idle_CPU => percent_mem(DIRECT) unchanged
- DIRECT is all non-subunit, non-node processes on the processor,
- (system and user) BUT... percent_CPU may neglect some system
- processes, such as screen saver, etc.

- These are the SPEC specifications for node.
- It is modeled as a machine, with respective state information,
- stimulus, and response.

MACHINE node

INHERIT period
STATE (s = [busy | idle], check_phase : time)
INVARIANT true
INITIALLY s = (busy, check_phase)

MESSAGE idle_check

access log_info
access p_info(direct)
WHEN users = 0
access p_info(direct)
WHEN percent_cpu = 0 and rem((time-check_phase), phase(busy_check))=0
SEND busy_check TO node
SEND node_idle to node_process
TRANSITION s = (idle, time)
OTHERWISE
SEND idle_check TO node
OTHERWISE
SEND idle_check TO node

MESSAGE busy_check

access log_info
access p_info(direct)
WHEN [(users > 0) | (percent_cpu > 0)] and
rem((time-check_phase), phase(idle_check))=0
SEND idle_check TO node
SEND node_busy to node_process
TRANSITION s = (busy, time)
WHEN (subunit_assigned & p_info(subunit) > 0)
OTHERWISE
SEND busy_check TO node

END

-- machine specification for node_process

MACHINE node_process

STATE s=(unassigned, running, asleep, completed)

INVARIANT true

INITIALLY s = unassigned

MESSAGE initialize

SEND idle_check TO node

-- initialize node state variables, also called init_node in
-- diagram.

MESSAGE node_busy

WHEN s = running

SEND sleep TO subunit

SEND unavail TO assign_node

TRANSITION s = asleep

WHEN s = unassigned

SEND unavail TO assign_node

OTHERWISE

SEND unavail TO assign_node

MESSAGE node_idle

WHEN s = asleep

SEND wake TO subunit

SEND avail TO assign_node

TRANSITION s = running

WHEN s = running

SEND avail TO assign_node

OTHERWISE

SEND avail TO assign_node

MESSAGE init_subunit(r,w)

-- Check state here and transition to assigned

SEND start(r,w) TO subunit

MESSAGE term_subunit

-- Received from assign_node, after detection that

-- this node's subunit has been lost from computation.

-- Reports from this process will be irrelevant.

-- Check state here and transition to unassigned

SEND term TO subunit

TRANSITION s = unassigned

MESSAGE state(s)

SEND state(s) TO assign_node

MESSAGE completed

-- received from subunit.

SEND avail TO assign_node

TRANSITION s = completed

```

MESSAGE ack_completed
-- received from assign_node
  TRANSITION s = unassigned

END

-- machine specification for subunit

MACHINE subunit
  STATE s = (run, sleep, complete) -- also, task_specific info
  INVARIANT true
  INITIALLY s = run

  MESSAGE start(r,w)
    SEND report to subunit

  MESSAGE sleep
    SEND report to subunit
    TRANSITION s = sleep

  MESSAGE wake
    TRANSITION s = run

  MESSAGE term
    SEND report to subunit
    TRANSITION s=complete
    SEND completed TO node_process

  MESSAGE report
    -- this is a temporal message. reports are made
    -- on a periodic basis to node_process.
    SEND state(s) to node_process

END

-- Note: node_process will have tight control on the subunit.
-- A signal to sleep will mean that the subunit will immediately cease
-- computation, and optionally make a report, before sleeping.
-- Computation in progress will not be completed.

-- machine specification for assign_node

MACHINE assign_node(j : job)

  STATE (j: sequence(s:subunit))
  INVARIANT : true
  INITIALLY : j ~=[ ]

  MESSAGE avail

```

```

FOR i = 1 TO count(s IN j)
  c = j(i)
  WHEN ~(c.active) & (c.mem <= avail.mem)
    REPLY init_subunit(c.id,r,w) TO node_process
    -- Also initialize timing logic for each subunit at this point.
    -- Should timer elapse before exec_state information is received,
    -- that subunit is considered "lost".
  END LOOP

MESSAGE unavail
  -- if this is received from a node_process currently
  -- assigned a subunit, this will invalidate the "loss" timer
  -- mentioned above, but will start a new timer, to
  -- guard against excessively long unavailability.

MESSAGE state(s)
  WHEN completed(s.completion_criteria)
    REPLY ack_complete

  WHEN period(loss_timer)
    REPLY term_subunit

  WHEN period(too_long)
    REPLY term_subunit

END

```

APPENDIX C. Source Code for Node Process

```
/*
 *
 *      NODE PROCESS
 *      a component of CEE.
 *      Supervise long running jobs.
 */

#ifndef lint
static char rcsid[] =
"$Header: /cm/src/uci/usr/local/polite/RCS/polite.c,v 1.2 88/06/16 14:01:19 sources Exp Locker: sources $
UCI-ICS Support";
#endif
```

```
/*
 * Revision 1.0, 3 September 1993.
 * Based on the original Polite program, source provided by thesis advisor.
 * Changes made include:
 * Input now taken from a file vice command line args, to include
 * original switches, as well as read and write resource files for client process.
 * Procedure now calls gethostname(), uses string result to find and read correct
 * input line in the pending subunits file.
 * Procedure now checks for user activity on local machine, writes availability to
 * file, for use by assign node.
 * Procedure immediately fork()'s to the background after initialization.
 * Procedure will wait until a subunit is assigned, then proceed with execution.
 * After execution, waiting resumes.
 * Read and write resources of client process are mapped to file descriptors 5 and 6,
 * respectively. These may be changed to stdin and stdout of client process.
 */
```

```
/* INCLUDES */
```

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include <utmp.h>
#include <signal.h>
#include <errno.h>
#include <pwd.h>
#include <string.h>
#include <ctype.h>
#include <varargs.h>
```

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
/* CONSTANTS */
```

```
#define UTMPNAME "/etc/utmp"
```

```
#define WTMPNAME "/usr/adm/wtmp"
```

```
#define MAXINT ((int)(((unsigned)-0)>>1))
```

```
struct utmp dummy;
```

```
#define UTMP_SIZE (sizeof(dummy))
```

```
#define TMPNAMELEN (sizeof(dummy.ut_name))
```

```
#define TMPLINELEN (sizeof(dummy.ut_line))
```

```
#define SUBUNITNAME "/work/busmiret/my_c/pending.subunits"
```

```
#define READIT "r"
```

```
#define WRITEIT "w"
```

```
#define ADDTOIT "a"
```

```
#define AVAILFILE "node_status."
```

```
#define MAXARGS 127
```

```
#define MAXTEST 4096
```

```
/* #define DEBUG dummy; */
```

```
/* TYPES */
```

```
typedef enum {FALSE, TRUE} Boolean;
```

```
#define NameList(name_size, num_names) \
```

```
    struct { \
```

```
        int len; \
```

```
        char names[num_names][name_size]; \
```

```
    } \
```

```
typedef enum {
```

```
    UNASSIGNED, RUNNING, ASLEEP, COMPLETED} node_state;
```

```
/* GLOBAL VARS */
```

```
Boolean waitforme; /* Stop process while calling user is on? */
```

```
int run_sleep, /* How long to sleep while job is running */
```

```
stop_sleep, /* How long to sleep while job is stopped */
```

```
nicelevel, /* Nice level increment for job */
```

```
maxtimeon, /* Maximum time a user can be on */
```

```
pgroup, /* Process group of job */
```

```
utmpfd, /* Utmp file */
```

```
wtmpfd, /* Wtmp file */
```

```
subunitfd; /* Subunit file */
```

```
NameList(TMPNAMELEN,200) wtmp_users; /* List of users found in wtmp */
```

```
NameList(TMPNAMELEN,200) ignore; /* Names of people to ignore if on */
```

```
FILE *fptin;
```

```
FILE *fptout;
```

```

FILE *ceewfp, *ceerfp;
char inbuffer[256];
Boolean avail;
node_state node_status;
int argc;
char *argv[MAXARGS];
char *cee_read, *cee_write;
char *machine_name[64];
int machine_name_len;
char *avail_file;

```

/* DECLARATIONS */

```

extern char *getenv();
extern long lseek();
Boolean utmp();
time_t time();

```

/* MACROS */

```

#define max(x,y) ((x) > (y) ? (x) : (y))
#define bool_char(i) ((i) ? 'T' : 'F')

```

```

#ifdef DEBUG
# define debug(x) debug_func x
#else
# define debug(x)
#endif

```

```

/* Database operations */
#define init(list) (list.len = 0)
#define find(name, list) (find_name(name, &list, sizeof(list.names[0])))
#define del(ptr, list) (strncpy(ptr, list.names[--list.len], \
                                sizeof(list.names[0])))
#define add(name, list) (strncpy(list.names[list.len++], name, \
                                sizeof(list.names[0])))

```

/* FIND_NAME :

Do a linear search for a name in a NameList.

```

*/
char *find_name(name, list, name_size)
    register char *name;
    register name_size;
    NameList(500,500) *list;
{
    register char *p, *end_names;

```

```

register char *names;

names = list->names[0];
end_names = names + name_size*list->len;

for (p = names; p < end_names; p += name_size) {
if (!strcmp(p, name, name_size)) {
    return p;
}
}
return NULL;
} /* end find_name */

/* OTHERS_ON:
Check to see if non-ignored users are on and return a boolean value.
*/
Boolean others_on()
{
    static struct stat utmp_stat;    /* Previous utmp status */
    static struct stat wtmp_stat;    /* Previous wtmp status */
    static Boolean retval = TRUE;    /* Previous return value */

    Boolean utmp_changed, wtmp_changed; /* Have [wu]tmp changed? */
    Boolean wtmp_shrunk;                /* Has wtmp shrunk? */
    struct stat stat_buf;                /* Temp status */

    debug(("Beginning others_on check"));

    /* Stat utmp */
    if (fstat(utmpfd, &stat_buf) == -1) {
pperror(UTMPNAME);
return retval;
    }

    /* Find out if utmp has been modified */
    utmp_changed = (Boolean)(stat_buf.st_mtime > utmp_stat.st_mtime);

    /* Copy new stat into saved status */
    bcopy((char*)&stat_buf, (char*)&utmp_stat, sizeof(stat_buf));

    /* Stat wtmp */
    if (fstat(wtmpfd, &stat_buf) == -1) {
pperror(WTMPNAME);
return retval;
    }

    /* Find out if wtmp has been modified */
    wtmp_changed = (Boolean)(stat_buf.st_mtime > wtmp_stat.st_mtime);

```

```

/* Find out if wtmp has shrunk */
wtmp_shrunk = (Boolean)(stat_buf.st_size < wtmp_stat.st_size);

/* Copy new stat into saved status */
bcopy((char*)&stat_buf, (char*)&wtmp_stat, sizeof(stat_buf));

debug(("utmp_changed is %c, wtmp_changed is %c, wtmp_shrunk is %c",
bool_char(utmp_changed), bool_char(wtmp_changed),
bool_char(wtmp_shrunk)));

/* If files have changed, re-read them */
if (utmp_changed || wtmp_changed) {
if (wtmp_changed)
    wtmp(wtmpfd, wtmp_shrunk);
return retval = utmp(utmpfd);
}

/* No changes, return previous value */
return retval;

} /* end others_on */

/* W T M P :
* Read the wtmp file and put any logged in users found there into wtmp_users.
*/
wtmp(fd, shrunk)
    Boolean shrunk;
{
    static long last_needed = 0L; /* Last place in file we need to read */

    NameList(TMPLINELEN, 200) ttys;
    struct utmp entries [100];
    long prev_pos, cur_pos, eof_pos;
    time_t oldest_time;
    register int n;

    /* Find the oldest time entry that won't be ignored */
    oldest_time = time((long*)0) - maxtimeon*60L;

    /* If file has shrunk, make no assumptions about where to stop reading */
    if (shrunk) last_needed = 0L;

    debug(("Checking %s", WTMPNAME));

    init(ttys);
    init(wtmp_users);
    cur_pos = eof_pos = lseek(fd, 0L, L_XTND);

    while (cur_pos > last_needed) {

```



```

prev_pos = cur_pos;
cur_pos = max(prev_pos - 100L*UTMP_SIZE, last_needed);

debug(("Seeking to pos %ld", cur_pos));

if (lseek(fd, cur_pos, L_SET) != cur_pos) {
    perror(WTMPNAME);
    return;
}
n = read(fd, (char*)entries, (int)(prev_pos - cur_pos)) / UTMP_SIZE;

debug((" %d entries read", n));

while (n-- > 0) {
    debug(("Got '%s' on line '%s'", entries[n].ut_name,
        entries[n].ut_line));
    if (entries[n].ut_line[0] == '~') { /* System reset */
        debug(("System reset"));
        last_needed = eof_pos;
        return;
    }
    else if (entries[n].ut_name[0] == 0) { /* Logout */
        /* Add to list of ttys that have logged out */
        if (find(entries[n].ut_line, ttys) == NULL)
            add(entries[n].ut_line, ttys);
        debug(("Logout"));
    }
    else if (nonuser(entries[n])) { /* Non-user */
        debug(("Non-user"));
    }
    else { /* Login */
        /* If no previous logout on tty, and do not ignore name */
        if (entries[n].ut_time >= oldest_time &&
            find(entries[n].ut_line, ttys) == NULL &&
            find(entries[n].ut_name, ignore) == NULL)
        {
            debug(("^^^ Non-ignored login ^^^, adding to users"));
            add(entries[n].ut_name, wtmp_users);
        }
        else {
            debug(("Ignored login"));
        }
    }
}
last_needed = eof_pos;
} /* end wtmp */

```

/* UTMP:

```

* Read the utmp file and if users found there are also in wtmp_users.
* return TRUE.
*/

```

```

Boolean utmp(fd)

```

```

    register fd;

```

```

{

```

```

    struct utmp entry;    /* One utmp entry */

```

```

    time_t oldest_time;   /* Oldest entry time that won't be ignored */

```

```

    debug(("Checking %s", UTMPNAME));

```

```

    /* Find the oldest time entry that won't be ignored */

```

```

    oldest_time = time((long*)0) - maxtimeon*60L;

```

```

    lseek(fd, 0L, L_SET);

```

```

    while (read(fd, (char *)&entry, UTMP_SIZE) == UTMP_SIZE) {

```

```

        debug(("Got '%s' on line '%s'", entry.ut_name, entry.ut_line));

```

```

        /* If found a login name that is not ignored */

```

```

        if (entry.ut_name[0] && !nonuser(entry) &&

```

```

            entry.ut_time >= oldest_time &&

```

```

            find(entry.ut_name, wtmp_users) != NULL)

```

```

        {

```

```

            debug(("Non-ignored user, utmp returning T"));

```

```

            return TRUE;

```

```

        }

```

```

    }

```

```

    debug(("Done reading utmp, returning F"));

```

```

    return FALSE;

```

```

} /* end utmp */

```

```

/* RECEIVE SUBUNIT:

```

```

* Check for the existence of an assignment file, search the file for the

```

```

* first available subunit, return the command for execution.

```

```

*/

```

```

receive_subunit()

```

```

{

```

```

    char *tempstr, *tempstr1;

```

```

    int p;

```

```

    Boolean found_my_line;

```

```

    typedef enum {

```

```

    RUN_SLEEP, STOP_SLEEP, WAITFORME, NICELEVEL, MAXTIMEON, IGNORE, PGROUP,

```

```

    READ_RES, WRITE_RES

```

```

    } arg_type;

```

```

#define NUMARGTYPE 9

```

```

    struct args_struct {

```

```

arg_type type;
char name[3];
    Boolean have_parm;
};
static struct args_struct args[NUMARGTYPE] = (
    (RUN_SLEEP, "rs", TRUE), (STOP_SLEEP, "ss", TRUE),
    (WAITFORME, "me", FALSE), (NICELEVEL, "n", TRUE),
    (MAXTIMEON, "mt", TRUE), (IGNORE, "ig", TRUE),
    (PGROUP, "pg", TRUE), (READ_RES, "r", TRUE),
    (WRITE_RES, "wr", TRUE)
);

register j, arglen, i;
char *parm;

found_my_line = FALSE;

/* Open the subunit.available file, and
   read in a single line, and close the file */

fptin = fopen(SUBUNITNAME, READIT);
while (!found_my_line){
    tempstr = fgets(inbuffer, 256, fptin);
    argc = 0;

    /* Build argv and count to argc by
       breaking the line read into strings,
       separated by spaces. */
    for(;;){
        tempstr = strtok(tempstr, " ");

        if(tempstr == NULL){
            found_my_line = TRUE;
            debug(("tempstr is found to be null, breaking."));
            break;
        }
        if (strcmp(tempstr, machine_name, machine_name_len) != 0 && argc == 0) {
            debug(("contents of tempstr not equal to machine_name, breaking"));
            break;
        }
    }
    else {
        found_my_line = TRUE;

        /* clear up any new_lines */
        tempstr1 = strchr(tempstr, '\n');
        if (tempstr1 != NULL) {
            debug(("This argument has a newline... removing it..."));
            p = strlen(tempstr);
            strcpy(tempstr1, "");
            strcat(tempstr1, tempstr, p-1);
        }
    }
}

```

```

    else {
        tempstr1 = tempstr;
    }
    argv[argc++] = strdup(tempstr1);
    debug(("argv[%d] is %s", argc-1, argv[argc-1]));
}
tempstr = NULL;
}
}
fclose(fptin);

/* Initialize argv[0] and argv[argc] */
argv[argc] = NULL;

/* Step through command line args */
for (i = 1; i < argc; ++i) {

    /* If arg not an option, quit */
    if (argv[i][0] != '-') {
        break;
    }

    /* Search for option in args table */
    for (j = 0; j < NUMARGTYPE; ++j) {
        arglen = strlen(args[j].name);
        if (!strcmp(argv[i]+1, args[j].name)) {
            break;
        }
    }

    /* Quit if this option not in table */
    if (j == NUMARGTYPE) {
        errmsg("Unknown option %s\n", argv[i]);
        exit(1);
    }

    /* If option has a parameter, retrieve it from end of current arg, or
     * from next argument.
     */
    if (args[j].have_parm) {
        parm = argv[i]+1+arglen;
        if (*parm == '\0') {
            parm = argv[++i];
            if (i == argc || argv[i][0] == '-') {
                errmsg("Need a parameter for %s option\n", args[j].name);
                exit(1);
            }
        }
    }
}

/* Branch out on argument type */

```

```

switch(args[j].type) {

    case RUN_SLEEP:
        run_sleep = myatoi(parm, "-rs", 10, 600);
        break;

    case STOP_SLEEP:
        stop_sleep = myatoi(parm, "-ss", 10, 600);
        break;

    case WAITFORME:
        waitforme = TRUE;
        break;

    case NICELEVEL:
        nicelevel = myatoi(parm, "-n", -40, 40);
        break;

    case MAXTIMEON:
        maxtimeon = myatoi(parm, "-mt", 30, MAXINT);
        break;

    case IGNORE:
        add(parm, ignore);
        break;

    case PGROUP:
        pgroup = myatoi(parm, "-pg", 0, 30000);
        break;
    case READ_RES:
        /* marry up parm to a fd or fp var */
        cee_read = parm;
        break;
    case WRITE_RES:
        /* marry up parm to a fd or fp var */
        cee_write = parm;
        break;
    )
}
debug(("i is %d", i));

return i;
} /* end receive_subunit */

/* NOTIFY_ASSIGN_NODE:
 * leave message for assign node whether the node is available to
 * accept a subunit or not. Writes "available" or "unavailable"
 * to the file notify.avail. When assign_node is completed, this
 * procedure will signal assign_node, rather than write to a file.

```

```

*/
notify_assign_node(proc_id)
{
    char *tempstr1, *tempstr2;

    /* set the output string */
    if (avail){
        tempstr1 = strcat("available", " ");
    }
    else{
        tempstr1 = strcat("unavailable", " ");
    }
    switch(node_status) {

        case UNASSIGNED:
            tempstr2 = strcat("not_assigned", "\n");
            break;
        case RUNNING:
            tempstr2 = strcat("running", "\n");
            break;
        case ASLEEP:
            tempstr2 = strcat("sleeping", "\n");
            break;
        case COMPLETED:
            tempstr2 = strcat("completed", "\n");
            break;
    }

    /* open the file, write message, and close */
    fptout = fopen(avail_file, WRITEIT);

    fprintf(fptout, "%s ", machine_name);
    fprintf(fptout, "%d ", proc_id);
    fputs(tempstr1, fptout);
    fputs(tempstr2, fptout);
    fclose(fptout);

} /* end notify_assign_node */


/* MYatoi:
 * Convert string to a number, but print an error and quit if number is out
 * of bounds.
 */
myatoi(numstr, errmsg, lo_bound, hi_bound)
char *numstr, *errmsg;
{
    int i = atoi(numstr);

    if (i < lo_bound || i > hi_bound) {

```

```

    errmsg("%s: value must be between %d and %d\n", errmsg, lo_bound,
        hi_bound);
    exit(1);
}
return i;
} /* end myatoi */

```

/* P P E R R O R :

- * Print out an error message appropriate to the error number in the
- * variable errno.

*/

```

pperror(s)
    register char *s;
{
    fputs("node_p: ", stderr);
    perror(s);
    fflush(stderr);
} /* end pperor */

```

/* E R R M S G :

Print out a nice error message to stderr, same arg conventions as printf.

```

*/
errmsg(va_alist)
    va_dcl
{
    va_list args;
    register char *fmt;

    fputs("node_p: ", stderr);
    va_start(args);
    fmt = va_arg(args, char *);
    vfprintf(stderr, fmt, args);
    va_end(args);
    fflush(stderr);
} /* end errmsg */

```

/* D E B U G _ F U N C :

Print out a message for debugging purpose, argument conventions are the same as printf.

```

*/
#ifdef DEBUG
debug_func(va_alist)
    va_dcl
{
    va_list args;

```

```

    register char *fmt;

    fputs("debug: ", stderr);
    va_start(args);
    fmt = va_arg(args, char *);
    vfprintf(stderr, fmt, args);
    va_end(args);
    putc('\n', stderr);
    fflush(stderr);
} /* end debug */
#endif

/* MAIN:
 * Initialize, parse up the args, start the job and enter the main loop.
 */
main()
{
    int cmdarg, child_id, z;
    int p_cfd[2], c_pfd[2];
    char *line1, *line2;
    char line[MAXTEST];
    int linelen;
    int stat_child;

    /* Open up the tmp files */
    utmpfd = open(UTMPNAME, O_RDONLY);
    wtmpfd = open(WTMPNAME, O_RDONLY);
    if (utmpfd == -1 || wtmpfd == -1) {
        perror(utmpfd == -1 ? UTMPNAME : WTMPNAME);
        exit(1);
    }

    /*check for the existence of a subunit job file*/
    subunitfd = open(SUBUNITNAME, O_RDONLY);
    if (subunitfd == -1){
        perror(SUBUNITNAME);
        exit(1);
    }
    close(subunitfd);

    machine_name_len = 64;
    if (gethostname(machine_name, machine_name_len) == 0){
        debug(("Hostname = %s\n", machine_name));
    }

    avail_file = AVAILFILE;
    strcat(avail_file, machine_name);

```



```

/* Fork off into the background */
switch(fork()) {
    case -1:          /* Error */
        perror("fork");
        exit(1);
    case 0:           /* Child */
        break;
    default:          /* Parent */
        exit(0);
}

```

```

for(;;){

```

```

    /* Initialize vars */

```

```

    init(ignore);
    run_sleep = 10;
    stop_sleep = 30;
    nicelevel = 0;
    maxtimeon = 540;
    pgroup = -1;
    waitforme = FALSE;
    node_status = UNASSIGNED;

```

```

    /* Add caller to ignore list if necessary */

```

```

    if (!waitforme) {

```

```

        char *uname;
        struct passwd *pwent;

```

```

        if ((pwent = getpwuid(getuid())) != NULL) {

```

```

            uname = pwent->pw_name;

```

```

        } else {

```

```

            uname = getenv("USER");

```

```

        }

```

```

        if (uname == NULL) {

```

```

            errmsg("Cannot determine user name\n");

```

```

            exit(1);

```

```

        }

```

```

        add(uname, ignore);

```

```

    }

```

```

/* check for non-ignored users on this node. Notify assign_node

```

```

 * of availability of this node. If unavailable, go to sleep,

```

```

 * then loop around. If not found, break loop, continue.

```

```

 */

```

```

for(;;){

```

```

    avail = !others_on();

```

```

    notify_assign_node(getpid());

```

```

    if (avail) {
        break;
    } else {
        sleep((unsigned)(stop_sleep));
    }
}

/* Parse up the command line */
cmdarg = receive_subunit();

debug(("cmdarg is %d", cmdarg));
debug(("waitforme is %c", bool_char(waitforme)));
debug(("run_sleep is %d", run_sleep));
debug(("stop_sleep is %d", stop_sleep));
debug(("nicelevel is %d", nicelevel));
debug(("maxtimeon is %d", maxtimeon));
debug(("pgroup is %d", pgroup));
# ifdef DEBUG
{
    int i;

    for (i = 0; i < ignore.len; ++i) {
        debug(("ignoring %s", ignore.names[i]));
    }
}
# endif

/* Ignore signals */
signal(SIGHUP, SIG_IGN);
signal(SIGINT, SIG_IGN);
signal(SIGQUIT, SIG_IGN);

/* If there is a command to execute */
if (cmdarg < argc) {

    avail = !avail;
    node_status = RUNNING;
    notify_assign_node(getpid());

    /* Open the read resource */
    if ((ceerfp = fopen(cee_read, "r")) == NULL) {
        debug(("error opening read resource"));
    }

    /* Open the write resource */
    if ((ceewfp = fopen(cee_write, "a+")) == NULL) {
        debug(("error opening write resource"));
    }

    /* Go execute the command */

```

```

switch(child_id = fork()) {

    case -1:          /* Error */
        perror("fork");
        exit(1);

    case 0:           /* Child */

        debug(("I received %d for reading", fileno(cceerfp)));
        debug(("I received %d for writing", fileno(ccewfp)));

        /* Renice the child */
        if (nicelevel != 0) {
            if (nice(nicelevel) == -1) {
                errmsg("Cannot change nice level\n");
            }
        }

        /* The following code is used to map read and write resources
        * to the child process stdin and stdout
        */
        /*
        if (fileno(cceerfp) != STDIN_FILENO) {
            if (dup2(fileno(ccewfp), STDIN_FILENO) != STDIN_FILENO) {
                debug(("dup2 error to stdin"));
            }
            close(ccewfp);
        }

        if (fileno(ccewfp) != STDOUT_FILENO) {
            if (dup2(fileno(ccewfp), STDOUT_FILENO) != STDOUT_FILENO) {
                debug(("dup2 error to stdout"));
            }
            close(ccewfp);
        }
        */

        /* Stop ourself until parent sets group id */
        kill(getpid(), SIGSTOP);

        /* Execute the command */
        debug(("exec'ing %s", argv[cmdarg]));
        execvp(argv[cmdarg], argv + cmdarg);

        /* Error if exec returns */
        perror(argv[cmdarg]);
        exit(1);

    default:          /* Parent */
        /* If pgroup not yet selected */
        if (pgroup < 0) {

```

```

    pgroup = child_id;

    /* Search for an empty process group */
    while (killpg(pgroup, 0) != -1 || errno != ESRCH) {
        if (-pgroup == 0) {
            kill(child_id, SIGKILL);
            errmsg(
                "Can't find an empty process group, use -pg\n");
            exit(1);
        }
    }
    setpgid(child_id, pgroup);

    /* close(0); close(1); */

    debug(("Group id is %d", pgroup));

    /* Wait for child to stop itself */
    while (wait3((union wait *)0, WUNTRACED, (struct rusage *)0)
        != child_id);

    /* Allow child to continue */
    if (kill(child_id, SIGCONT) == -1) {
        perror("Can't restart child");
        exit(1);
    }

    close(fileno(ccerfp));
    close(fileno(ccewfp));
    break;
}
}

if (pgroup < 0) {
    /* errmsg("Must specify a command to execute or a process group\n"); */
    debug(("Must specify a command to execute or a process group\n"));

    /* exit(1); */
}
else {
    /*
    close(p_cfd[0]);
    close(c_pfd[1]);
    */

    /* Pause to allow child to exec */
    sleep(1);

    /* Main Loop */
    for (;;) {
        Boolean group_stopped;

```

```

group_stopped = others_on();
node_status = group_stopped ? ASLEEP : RUNNING;

if (killpg(pgroup, group_stopped ? SIGSTOP : SIGCONT) < 0
    && errno == ESRCH)
{
    avail = !avail;
    node_status = COMPLETED;
    notify_assign_node(getpid());
    debug(("Process group exited, polite exiting normally"));

    if (child_id > 0){

        /* Wait for child to stop itself */
        while (wait(&stat_child) != child_id);
        break;
    }
}
notify_assign_node(getpid());

debug(("Process %s", group_stopped ? "stopped" : "running"));

sleep((unsigned)(group_stopped ? stop_sleep : run_sleep));
}
/*
close(ccerfp);
close(ccewfp);
*/
sleep((unsigned)run_sleep);
} /* end main */

```

LIST OF REFERENCES

- [BEGU93] Beguelin, A., Dongarra, J.J., Geist, G.A., Jiang, W., Manchek, R., Moore, K., and Sunderam, V.S., *The PVM Project*, Emory University, 1993.
- [BERZ88] Berzins, Valdis and Luqi, *Software Engineering with Abstractions*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
- [BERZ89] Berzins, Valdis and Kopas, Robert, *A Student's Guide to Spec*, Department of Computer Science, Naval Postgraduate School, Monterey, California, 1989.
- [BSDM89] BSD online manual page, Solbourne Computer Inc., 1989.
- [DONG93] Dongarra, Jack J., Geist, G.A., Manchek, Robert, and Sunderam, V.S., *Integrated PVM Framework Supports Heterogeneous Network Computing*, Emory University, January 3, 1993.
- [GELE82] Gelernter, David Hillel, *An Integrated Microcomputer Network for Experiments in Distributed Programming*, PhD Dissertation, State University of New York at Stony Brook, 1982.
- [GELE88] Gelernter, David Hillel, "Getting the Job Done", *Byte*, Vol. 13, No. 2, pp. 301-308, November 1988.
- [HENN91] Hennessy, J.L. and Jouppi, N.P., "Computer Technology and Architecture: An Evolving Interaction", *IEEE Computer*, Vol.24, No.9, pp.18-29, September 1991.
- [LELE90] Leler, Wm., "Linda Meets Unix", *IEEE Computer*, Vol. 23, No. 2, pp. 43-54, February 1990.
- [MARK92] Markoff, John, "David Gelernter's Romance with Linda", *The New York Times*, pp. F1-F6, January 19, 1992.
- [PFLE89] Pfleeger, Charles P., *Security in Computing*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [POLI88] Polite online manual page, unpublished manuscript, University of California, Irvine, Information and Computer Science Department, 1988.
- [POLS88] Polite source code, unpublished manuscript, University of California, Irvine, Information and Computer Science Department, 1988.
- [PSMA89] PS online manual page, Solbourne Computer, Inc., 1989.

[WHIT88] Whiteside, Robert A. and Leichter, Jerrold S., "Using Linda for Supercomputing On a Local Area Network", *Supercomputing '88*, IEEE Computer Society, pp. 192-199.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5002	2
Director, Training and Education MCCDC, Code C46 1019 Elliot Road Quantico, VA 22134-5027	1
Gordon Schacher Dean of Faculty and Graduate Studies Code 07 Naval Postgraduate School Monterey, CA 93943-5100	1
Ted Lewis, Professor and Chairman Department of Computer Science Code CSLt Naval Postgraduate School Monterey, CA 93943-5118	1
Computer Technology Programs Code 37 Naval Postgraduate School Monterey, CA 93943-5119	1
Prof. Timothy J. Shimeall Computer Science Department Code CS/Sm Naval Postgraduate School Monterey, CA 93943-5118	3
Mr. Roger Stemp Computer Science Department Code CS/Sp Naval Postgraduate School Monterey, CA 93943-5118	2

CAPT Terence E. Busmire
2067 Kendall Rd.
Kendall, NY 14476

2